

Community Experience Distilled

Mastering ElasticSearch

Extend your knowledge on ElasticSearch, and querying and data handling, along with its internal workings

Rafał Kuć
Marek Rogoziński

[PACKT] open source*
PUBLISHING

目錄

Mastering Elasticsearch(中文版)

译者：帅广应

GitHub：https://github.com/shgy/mastering-elasticsearch.git

邮箱：810050504@qq.com

微博：@帅广应s

说明：学习Elasticsearch以及了解一下gitbook这个平台

第1章 认识Elasticsearch

希望读者通过阅读本书，能够扩展和巩固ElasticSearch的基础知识。假定读者已经知道用单个请求(curl)和批量索引向ElasticSearch导入数据；也知道如何发送请求获取目标文档；也知道如何通过filter过滤查询结果。使结果更精确；也知道如何使用facet/aggregation机制来对结果进行统计处理，在学习ElasticSearch那些激动人心的功能之前，还是需要快速了解一下Apache Lucene。Apache Lucene，一种全文检索工具。ElasticSearch就是构建在Lucene之上的。与此同时，ElasticSearch也沿袭了Lucene的基本概念。如果想更快地理解ElasticSearch,就必须牢记Lucene的基本概念。当然，记住概念是很简单的。 但是如果想掌握Elasticsearch,在记住Lucene概念概念的基础之上，还必须理解这些概念。在本章，我们将学到如下的知识。

- Apache Lucene的简单介绍
- Lucene的总体架构
- 文本解析(analysis)的过程
- ElasticSearch的基本概念
- ElasticSearch的内部通信机制

认识Apache Lucene

为了更深入地理解ElasticSearch的工作原理，特别是索引和查询这两个过程，理解Lucene的工作原理至关重要。本质上，ElasticSearch是用Lucene来实现索引的查询功能的。如果读者没有用过Lucene，下面的几个部分将为您介绍Lucene的基本概念。

熟悉Lucene

读者也许会产生疑问，为什么ElasticSearch 的创造者最终采用Lucene而不是自己开发相应功能的组件。我们也不知道为什么，因为我们不是决策者。但是我们可以猜想可能是因为Lucene是一个成熟的、高性能的、可扩展的、轻量级的，而且功能强大的搜索引擎包。Lucene的核心jar包只有一个文件，而且不依赖任何第三方jar包。更重要的是，它提供的索引数据和检索数据的功能开箱即用。当然，Lucene也提供了多语言支持，具有拼写检查、高亮等功能；但是如果你不需要这些功能，你只需要下载Lucene的核心jar包，应用到你的项目中就可以了。

总体架构

尽管我希望直奔主题，介绍Lucene的架构，但是首先必须理解一些概念才能更好地理解Lucene的架构，这些概念是：

- **Document**: 它是在索引和搜索过程中数据的主要表现形式，或者称“载体”，承载着我们索引和搜索的数据,它由一个或者多个域(Field)组成。
- **Field**: 它是Document的组成部分，由两部分组成，名称(name)和值(value)。
- **Term**: 它是搜索的基本单位，其表现形式为文本中的一个词。
- **Token**: 它是单个Term在所属Field中文本的呈现形式，包含了Term内容、Term类型、Term在文本中的起始及偏移位置。

Apache Lucene把所有的信息都写入到一个称为**倒排索引**的数据结构中。这种数据结构把索引中的每个Term与相应的Document映射起来，这与关系型数据库存储数据的方式有很大的不同。读者可以把倒排索引想象成这样的一种数据结构：数据以Term为导向，而不是以Document为导向。下面看看一个简单的倒排索引是什么样的，假定我们的Document只有title域(Field)被编入索引，Document如下：

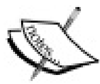
* ElasticSearch Servier (document 1) * Mastering ElasticSearch (document 2) * Apache Solr 4 Cookbook (document 3) 所以索引(以一种直观的形式)展现如下： | Term | count | Docs | | -- | -- | -- | | 4 | 1 | <3> | |Apache | 1 | <3> | | Cookbook | 1 | <3> | | ElasticSearch | 2 | <1> <2> | | Mastering | 1 | <1> | | Server | 1 | <1> | | Solr | 1 | <3> |

正如所看到的那样，每个词都指向它所在的文档号(Document Number/Document ID)。这样的存储方式使得高效的信息检索成为可能，比如基于词的检索(term-based query)。此外，每个词映射着一个数值(Count)，它代表着Term在文档集中出现的频繁程度。

当然，Lucene创建的真实索引远比上文复杂和先进。这是因为在Lucene中，词向量(由单独的一个Field形成的小型倒排索引，通过它能够获取这个特殊Field的所有Token信息)可以存储；所有Field的原始信息可以存储；删除Document的标记信息可以存储……。核心在于了解数据的组织方式，而非存储细节。

每个索引被分成了多个段(Segment)，段具有一次写入，多次读取的特点。只要形成了，段就无法被修改。例如：被删除文档的信息被存储到一个单独的文件，但是其它的段文件并没有被修改。

需要注意的是，多个段是可以合并的，这个合并的过程称为**segments merge**。经过强制合并或者Lucene的合并策略触发的合并操作后，原来的多个段就会被Lucene创建的更大的一个段所代替了。很显然，段合并的过程是一个I/O密集型的任务。这个过程会清理一些信息，比如会删除.del文件。除了精减文件数量，段合并还能够提高搜索的效率，毕竟同样的信息，在一个段中读取会比在多个段中读取要快得多。但是，由于段合并是I/O密集型任务，建议不好强制合并，小心地配置好合并策略就可以了。



如果想了解段由哪些文件组成，想了解每个文件中存储了什么信息，可以参考Apache Lucene documentation ,访问地址：http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/codecs/lucene45/package-summary.html.

分析你的文本

问题到这里就变得稍微复杂了一些。传入到Document中的数据是如何转变成倒排索引的？查询语句是如何转换成一个个Term使高效率文本搜索变得可行？这种转换数据的过程就称为文本分析(analysis)

文本分析工作由**analyzer**组件负责。analyzer由一个分词器(tokenizer)和0个或者多个过滤器(filter)组成,也可能会有0个或者多个字符映射器(character mappers)组成。

Lucene中的**tokenizer**用来把文本拆分成一个个的Token。Token包含了比较多的信息，比如Term在文本的中的位置及Term原始文本，以及Term的长度。文本经过**tokenizer**处理后的结果称为token stream。token stream其实就是一个个Token的顺序排列。token stream将等待着filter来处理。

除了**tokenizer**外，Lucene的另一个重要组成部分就是filter链，filter链将用来处理Token Stream中的每一个token。这些处理方式包括删除Token,改变Token，甚至添加新的Token。Lucene中内置了许多filter，读者也可以轻松地自己实现一个filter。有如下内置的filter：

- **Lowercase filter**：把所有token中的字符都变成小写
- **ASCII folding filter**：去除tonken中非ASCII码的部分
- **Synonyms filter**：根据同义词替换规则替换相应的token
- **Multiple language-stemming filters**：把Token(实际上是Token的文本内容)转化成词根或者词干的形式。

所以通过Filter可以让analyzer有几乎无限的处理能力：因为新的需求添加新的Filter就可以了。

索引和查询

在我们用Lucene实现搜索功能时，也许会有读者不明觉历：上述的原理是如何对索引过程和搜索过程产生影响？

索引过程：Lucene用用户指定好的analyzer解析用户添加的Document。当然Document中不同的Field可以指定不同的analyzer。如果用户的Document中有`title`和`description`两个Field，那么这两个Field可以指定不同的analyzer。

搜索过程：用户的输入查询语句将被选定的查询解析器(query parser)所解析,生成多个Query对象。当然用户也可以选择不解析查询语句，使查询语句保留原始的状态。在ElasticSearch中，有的Query对象会被解析(analyzed)，有的不会，比如：前缀查询(prefix query)就不会被解析，精确匹配查询(match query)就会被解析。对用户来说，理解这一点至关重要。

对于索引过程和搜索过程的数据解析这一环节，我们需要把握的重点在于：倒排索引中词应该和查询语句中的词正确匹配。如果无法匹配，那么Lucene也不会返回我们喜闻乐见的结果。举个例子：如果在索引阶段对文本进行了转小写(lowercasing)和转变成词根形式(stemming)处理，那么查询语句也必须进行相同的处理，不然就搜索结果就会是竹篮打水——一场空。

Lucene查询语言

ElasticSearch提供的一些查询方式(query types)能够被Lucene的查询解析器(query parser)语法所支持。由于这个原因，我们来深入学习Lucene查询语言，了解其庐山真面目吧。

基础语法

用户使用Lucene进行查询操作时，输入的查询语句会被分解成一个或者多个Term以及逻辑运算符号。一个Term，在Lucene中可以是一个词，也可以是一个短语(用双引号括引来的多个词)。如果事先设定规则：解析查询语句，那么指定的analyzer就会用来处理查询语句的每个term形成Query对象。

一个Query对象中会存在多个布尔运算符，这些布尔运算符将多个Term关联起来形成查询子句。布尔运算符号有如下类型：

- `AND(与)`:给定两个Term(左运算对象和右运算对象)，形成一个查询表达式。只有两个Term都匹配成功，查询子句才匹配成功。比如：查询语句"apache AND lucene"的意思是匹配含apache且含lucene的文档。
- `OR(或)`:给定的多个Term，只要其中一个匹配成功，其形成的查询表达式就匹配成功。比如查询表达式"apache OR lucene"能够匹配包含“apache”的文档，也能匹配包含"lucene"的文档，还能匹配同时包含这两个Term的文档。
- `NOT(非)`: 这意味着对于与查询语句匹配的文档，NOT运算符后面的Term就不能在文档中出现的。例如：查询表达式“lucene NOT elasticsearch”就只能匹配包含lucene但是不含elasticsearch的文档。

此外，我们也许会用到如下的运算符：

- **+**这个符号表明：如果想要查询语句与文档匹配，那么给定的Term必须出现在文档中。例如：希望搜索到包含关键词lucene,最好能包含关键词apache的文档，可以用如下的查询表达式："+lucene apache"。
- **-**这个符号表明：如果想要查询语句与文档匹配，那么给定的Term不能出现在文档中。例如：希望搜索到包含关键词lucene,但是不含关键词elasticsearch的文档，可以用如下的查询表达式："+lucene -elasticsearch"。

如果在Term前没有指定运算符，那么默认使用OR运算符。
此外，也是最后一点：查询表达式可以用小括号组合起来，形成复杂的查询表达式。比如：

```
elasticsearch AND (mastering OR book)
```

多域查询

当然，跟ElasticSearch一样，Lucene中的所有数据都是存储在一个个的Field中，多个Field形成一个Document。如果希望查询指定的Field,就需要在查询表达式中指定Field Name(此域名非彼域名)，后面接一个冒号，紧接着一个查询表达式。例如：查询title域中包含关键词elasticsearch的文档，查询表达式如下：

```
title:elasticsearch
```

也可以把多个查询表达式用于一个域中。例如：查询title域中含关键词elasticsearch并且含短语“mastering book”的文档，查询表达式如下：

```
title:(+elasticsearch +"mastering book")
```

当然，也可以换一种写法，作用是一样的：

```
+title:elasticsearch +title:"mastering book")
```

词语修饰符

除了可以应用简单的关键词和查询表达式实现标准的域查询，Lucene还支持往查询表达式中传入修饰符使关键词具有变形能力。最常用的修饰符，也是大家都熟知的，就是通配符。Lucene支持?和*两种通配符。?可以匹配任意单个字符，而*能够匹配多个字符。

[请注意出于性能考虑，默认的通配符不能是关键词的首字母。]

此外，Lucene支持模糊查询(fuzzy query)和邻近查询(proximity query)。语法规则是查询表达式后面接一个~符号，后面紧跟一个整数。如果查询表达式是单独一个Term，这表示我们的搜索关键词可以由Term变形(替换一个字符，添加一个字符，删除一个字符)而来，即与Term是相似的。这种搜索方式称为模糊搜索(fuzzy search)。在~符号后面的整数表示最大编辑距离。例如：执行查询表达式 "writer~2"能够搜索到含writer和writers的文档。

当~符号用于一个短语时，~后面的整数表示短语中可接收的最大的词编辑距离(短语中替换一个词，添加一个词，删除一个词)。举个例子,查询表达式title:"mastering elasticsearch"只能匹配title域中含"mastering elasticsearch"的文档，而无法匹配含"mastering book elasticsearch"的文档。但是如果查询表达式变成title:"mastering elasticsearch"~2,那么两种文档就都能够成功匹配了。

此外，我们还可以使用加权(boosting)机制来改变关键词的重要程度。加权机制的语法是一个^符号后面接一个浮点数表示权重。如果权重小于1，就会降低关键词的重要程度。同理，如果权重大于1就会增加关键词的重要程度。默认的加权值为1。可以参考 第2章 活用用户查询语

言 的 *Lucene*默认打分规则详解 章节部分的内容来了解更多关于加权(boosting)是如何影响打分排序的。

除了上述的功能外，Lucene还支持区间查询(range searching),其语法是用中括号或者}表示区间。例如：如果我们查询一个数值域(numeric field)，可以用如下查询表达式：

```
price:[10.00 TO 15.00]
```

这条查询表达式能查询到price域的值在10.00到15.00之间的所有文档。

对于string类型的field，区间查询也同样适用。例如：

```
name:[Adam TO Adria]
```

这条查询表达式能查询到name域中含关键词Adam到关键词Adria之间关键词(字符串升序，且闭区间)的文档。

如果希望区间的边界值不会被搜索到，那么就需要用大括号替换原来的中括号。例如，查询price域中价格在10.00(10.00要能够被搜索到)到15.00(15.00不能被搜索到)之间的文档，就需要用如下的查询表达式：

```
price:[10.00 TO 15.00}
```

处理特殊字符

如果在搜索关键词中出现了如下字符集合中的任意一个字符，就需要用反斜杠(\\)进行转义。字符集合如下： +, -, &&, || , ! , (,) , { } , [ ] , ^, " , ~, *, ?, : , \, / 。例如，查询关键词 abc"efg 就需要转义成 abc\"efg。

认识 ElasticSearch

如果你已经在学习本书的知识点，就说明你可能或多或少知道一些ElasticSearch的相关知识，至少已经了解了其核心概念和基本用法了。为了更深入地理解这款搜索引擎的工作原理，还是简单的论述一下相关知识吧。

大家应该都已经知道ElasticSearch是一款企业应用型的软件工具，用来建立搜索相关的程序。它最初由ShayBanon编写(它的前身是compass)，并且于2010年二月份发布了第一个版本。在随后的几年中，ElasticSearch发展迅猛，并且成为了开源的商业技术解决方案家族中一一支重要的力量。ElasticSearch也跻身了开源项目下载Top榜，每月的下载量突破了200000次。

基本概念

让我们一起把ElasticSearch的基本概念和其特性浏览一遍。

索引(Index)

ElasticSearch把数据存放到一个或者多个索引(indices)中。如果用关系型数据库模型对比，索引(index)的地位与数据库实例(database)相当。索引存放和读取的基本单元是文档(Document)。我们也一再强调，ElasticSearch内部用Apache Lucene实现索引中数据的读写。读者应该清楚的是：在ElasticSearch中被视为单独的一个索引(index)，在Lucene中可能不止一个。这是因为在分布式体系中，ElasticSearch会用到分片(shards)和备份(replicas)机制将一个索引(index)存储多份。

文档(Document)

在ElasticSearch的世界中，文档(Document)是主要的存在实体(在Lucene中也是如此)。所有的ElasticSearch应用需求到最后都可以统一建模成一个检索模型：检索相关文档。文档(Document)由一个或者多个域(Field)组成，每个域(Field)由一个域名(此域名非彼域名)和一个或者多个值组成(有多个值的值称为多值域(multi-valued))。在ElasticSeach中，每个文档(Document)都可能会有不同的域(Field)集合；也就是说文档(Document)是没有固定的模式和统一的结构。文档(Document)之间保持结构的相似性即可(Lucene中的文档(Document)也秉持着相同的规定)。实际上，ElasticSearch中的文档(Document)就是Lucene中的文档(Document)。从客户端的角度来看，文档(Document)就是一个JSON对象(关于JSON格式的相关信息,请参看hhttp://en.wikipedia.org/wiki/JSON)。
参数映射(Mapping)
在 *1.1*节 认识*Apache Lucene* 中已经提到，所有的文档(Document)在存储之前都必须经过分析(analyze)流程。用户可以配置输入文本分解成Token的方式；哪些Token应该被过滤掉；或者其它的的处理流程，比如去除HTML标签。此外，ElasticSearch提供的各种特性，比如排序的相关信息。保存上述的配置信息，这就是参数映射(Mapping)在ElasticSearch中扮演的角色。尽管ElasticSearch可以根据域的值自动识别域的类型(field type)，在生产应用中，都是需要自己配置这些信息以避免一些奇的问题发生。要保证应用的可控性。

文档类型(Type)

每个文档在ElasticSearch中都必须设定它的类型。文档类型使得同一个索引中在存储结构不同文档时，只需要依据文档类型就可以找到对应的参数映射(Mapping)信息，方便文档的存取。

节点(Node)

单独一个ElasticSearch服务器实例称为一个节点。对于许多应用场景来说，部署一个单节点的ElasticSearch服务器就足够了。但是考虑到容错性和数据过载，配置多节点的ElasticSearch集群是明智的选择。

集群(Cluster)

集群是多个ElasticSearch节点的集合。这些节点齐心协力应对单个节点无法处理的搜索需求和数据存储需求。集群同时也是应对由于部分机器(节点)运行中断或者升级导致无法提供服务这一问题的利器。ElasticSearch提供的集群各个节点几乎是无缝连接(所谓无缝连接，即集群对外而言是一个整体，增加一个节点或者去掉一个节点对用户而言是透明的<个人理解，仅供参考>)。在ElasticSearch中配置一个集群非常简单，在我们看来，这是在与同类产品中竞争所体现出的最大优势。

分片索引(Shard)

前面已经提到，集群能够存储超出单机容量的信息。为了实现这种需求，ElasticSearch把数据分发到多个存储Lucene索引的物理机上。这些Lucene索引称为分片索引，这个分发的过程称为索引分片(Sharding)。在ElasticSearch集群中，索引分片(Sharding)是自动完成的，而且所有分片索引(Shard)是作为一个整体呈现给用户的。需要注意的是，尽管索引分片这个过程是自动的，但是在应用中需要事先调整好参数。因为集群中分片的数量需要在索引创建前配置好，而且服务器启动后是无法修改的，至少目前无法修改。

索引副本(Replica)

通过索引分片机制(Sharding)可以向ElasticSearch集群中导入超过单机容量的数据，客户端操作任意一个节点即可实现对集群数据的读写操作。当集群负载增长，用户搜索请求阻塞在单个节点上时，通过索引副本(Replica)机制就可以解决这个问题。索引副本(Replica)机制的的思路很简单：为索引分片创建一份新的拷贝，它可以像原来的主分片一样处理用户搜索请求。同时也顺便保证了数据的安全性。即如果主分片数据丢失，ElasticSearch通过索引副本使得数据不丢失。索引副本可以随时添加或者删除，所以用户可以在需要的时候动态调整其数量。

时间之门(Gateway)

在运行的过程中，ElasticSearch会收集集群的状态、索引的参数等信息。这些数据被存储在Gateway中。

ElasticSearch背后的核心理念

ElasticSearch是构建在极少数的几个概念之上的。ElasticSearch的开发团队希望它能够快速上手，可扩展性强。而且这些核心特性体现在ElasticSearch的各个方面。从架构的角度来看，这些主要特性是：

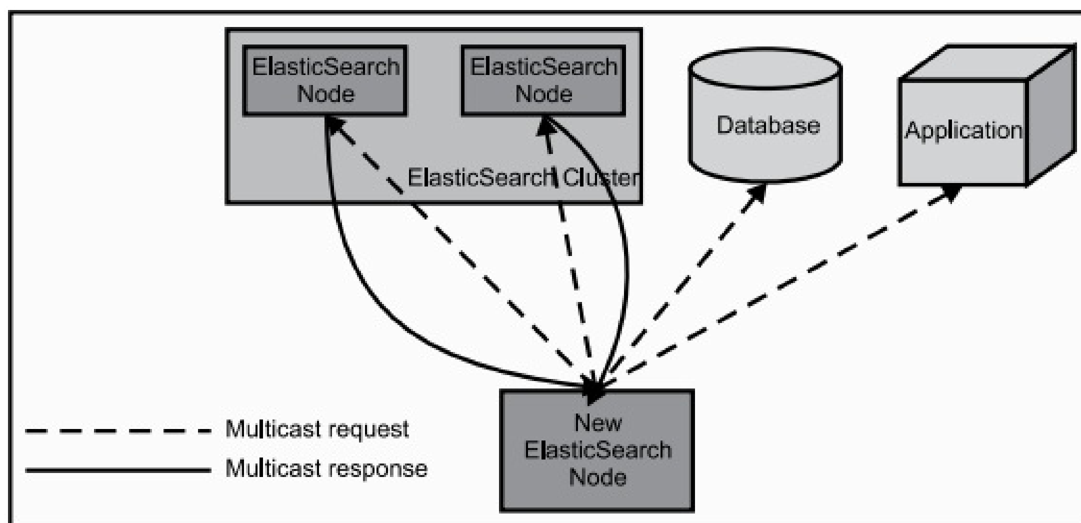
- 开箱即用。安装好ElasticSearch后，所有参数的默认值都自动进行了比较合理的设置，基本不需要额外的调整。包括内置的发现机制(比如Field类型的自动匹配)和自动化参数配置。
- 天生集群。ElasticSearch默认工作在集群模式下。节点都将视为集群的一部分，而且在启动的过程中自动连接到集群中。
- 自动容错。ElasticSearch通过P2P网络进行通信，这种工作方式消除了单点故障。节点自动连接到集群中的其它机器，自动进行数据交换及以节点之间相互监控。索引分片
- 扩展性强。无论是处理能力和数据容量上都可以通过一种简单的方式实现扩展，即增添新的节点。
- 近实时搜索和版本控制。由于ElasticSearch天生支持分布式，所以延迟和不同节点上数据的短暂性不一致无可避免。ElasticSearch通过版本控制(versioning)的机制尽量减少问题的出现。

ElasticSearch的工作原理

接下来简单了解一下ElasticSearch的工作原理。

启动过程

当ElasticSearch的节点启动后，它会利用多播(multicast)(或者单播，如果用户更改了配置)寻找集群中的其它节点，并与之建立连接。这个过程如下图所示



在集群中，一个节点被选举成主节点(master node)。这个节点负责管理集群的状态，当群集的拓扑结构改变时把索引分片分派到相应的节点上。



需要注意的是，从用户的角度来看，主节点在ElasticSearch中并没有占据着重要的地位，这与其它的系统(比如数据库系统)是不同的。实际上用户并不需要知道哪个节点是主节点；所有的操作需求可以分发到任意的节点，ElasticSearch内部会完成这些让用户感到不明觉历的工作。在必要的情况下，任何节点都可以并发地把查询子句分发到其它的节点，然后合并各个节点返回的查询结果。最后返回给用户一个完整的数据集。所有的这些工作都不需要经过主节点转发(节点之间通过P2P的方式通信)。

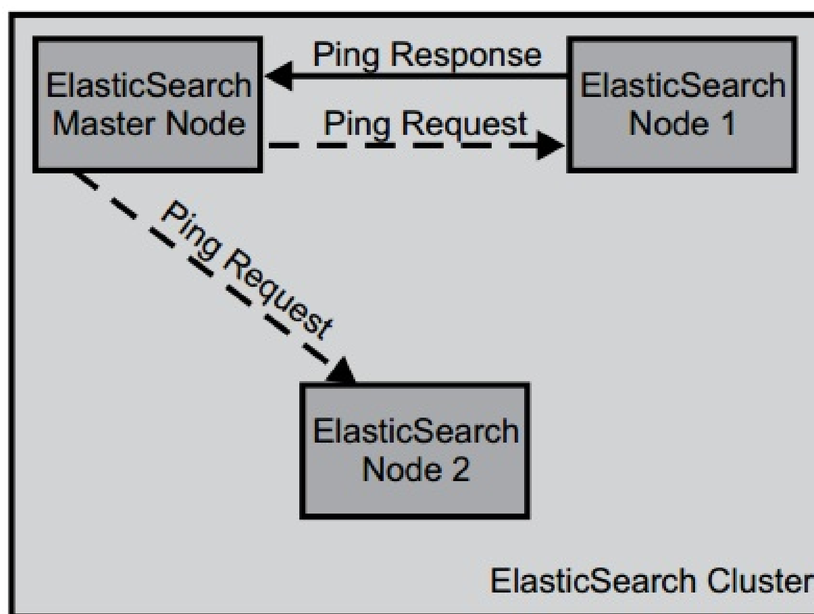
主节点会去读取集群状态信息，在必要的时候，会进行恢复工作。在这个阶段，主节点会去检查哪些分片可用，决定哪些分片作为主分片。处理完成后，集群就会转入到黄色状态。

这意味着集群已经可以处理搜索请求了，但是还没有火力全开(这主要是由于所有的主索引分片(primary shard)都已经分配好了，但是索引副本还没有)。接下来需要做的事情就是找到复制好的分片，并设置成索引副本。当一个分片的副本数量太少时，主节点会决定将缺少的分片放置到哪个节点中，并且依照主分片创建副本。所有工作完成后，集群就会变成绿色的状态(表示所有的主分片的索引副本都已经分配完成)。

探测失效节点

在正常工作时，主节点会监控所有的节点，查看各个节点是否工作正常。如果在指定的时间里面，节点无法访问，该节点就被视为出故障了，接下来错误处理程序就会启动。集群需要重新均衡——由于该节点出现故障，分配到该节点的索引分片丢失。其它节点上相应的分片就会把工作接管过来。换句话说，对于每个丢失的主分片，新的主分片将从剩余的分片副本(Replica)中选举出来。重新安置新的分片和副本的这个过程可以通过配置来满足用户需求。更多相关信息可以参看 第4章 分布式索引架构。

由于只是展示ElasticSearch的工作原理，我们就以下图三个节点的集群为例。集群中有一个主节点和两个数据节点。主节点向其它的节点发送Ping命令然后等待回应。如果没有得到回应(实际上可能得不到回复的Ping命令个数取决于用户配置)，该节点就会被移出集群。



与ElasticSearch进行通信

我们已经探讨了ElasticSearch是如何构建起来的，但是归根到底，最重要的是如何往ElasticSearch中添加数据以及如何查询数据。为了实现上述的需求，ElasticSearch提供了精心设计的API。这些主要的API都是基于REST风格(参看http://en.wikipedia.org/wiki/Pepresentational_state_transfer)。而且这些API非常容易与其它能够处理HTTP请求的系统进行集成。

ElasticSearch认为数据应该伴随在URL中，或者作为请求的主体(request body)，以一种JSON格式(http://en.wikipedia.org/wiki/JSON)的文档发送给服务器。如果读者用Java或者其它运行在JVM虚拟机上的语言，应该关注一下Java API，它除了是群集中内置的REST风格API外，功能与URL请求是一样的。

值得一提的是在ElasticSearch内部，节点之间的通信也是用相关的Java API。如果想了解关于Java API更多的内容，可以阅读 第8章 *ElasticSearch Java API* ，但是现在还是简要了解一下本章提供的一些API的功能和使用方法。注意本章仅仅是对相关知识的简单提点(作者会假定读者已经使对这些知识有所了解)。如果事先没有了解相关知识，强烈建议读者去学习一下。比如本书就覆盖了所有的知识点。

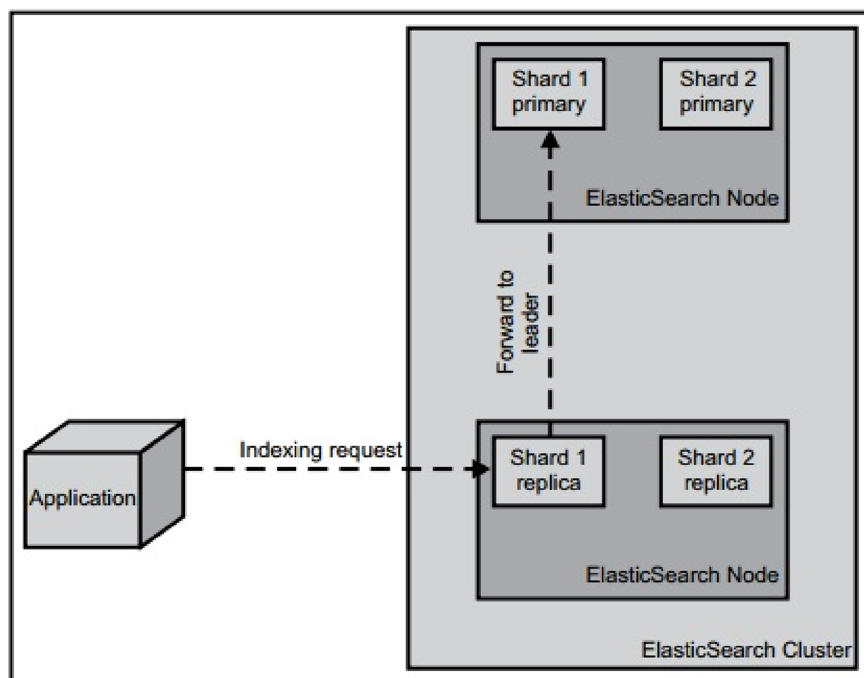
索引数据

ElasticSearch提供了4种索引数据的办法。最简单的是使用索引API,索引API。通过它可以将文档添加到指定的索引中去。比如，通过curl工具(访问http://curl.haxx.se/)，我们可以通过如下的命令创建一个新的文档：

```
curl -XPUT http://localhost:9200/blog/article/1 -d '{
    "title": "New version of Elastic Search released!", "content": "...",
"tags": ["announce", "elasticsearch", "release"]
}'
```

第2种和第3种办法可以通过bulk API和UDP bulk API批量添加文档。通常的bulk API采用HTTP协议，UDP bulk API采用非连接的数据包协议。UDP协议传输速度会更快，但是可靠性要差一点。最后一种办法就是通过river插件。river运行在ElasticSearch集群的节点上，能够从外部系统中获取数据。

有一点需要注意，索引数据的操作只会发生在主分片(primary shard)上，而不会发生在分片副本(Replica) 上。如果索引数据的请求发送到的节点没有合适的分片或者分片是副本，那么请求会被转发到含有主分片的节点。

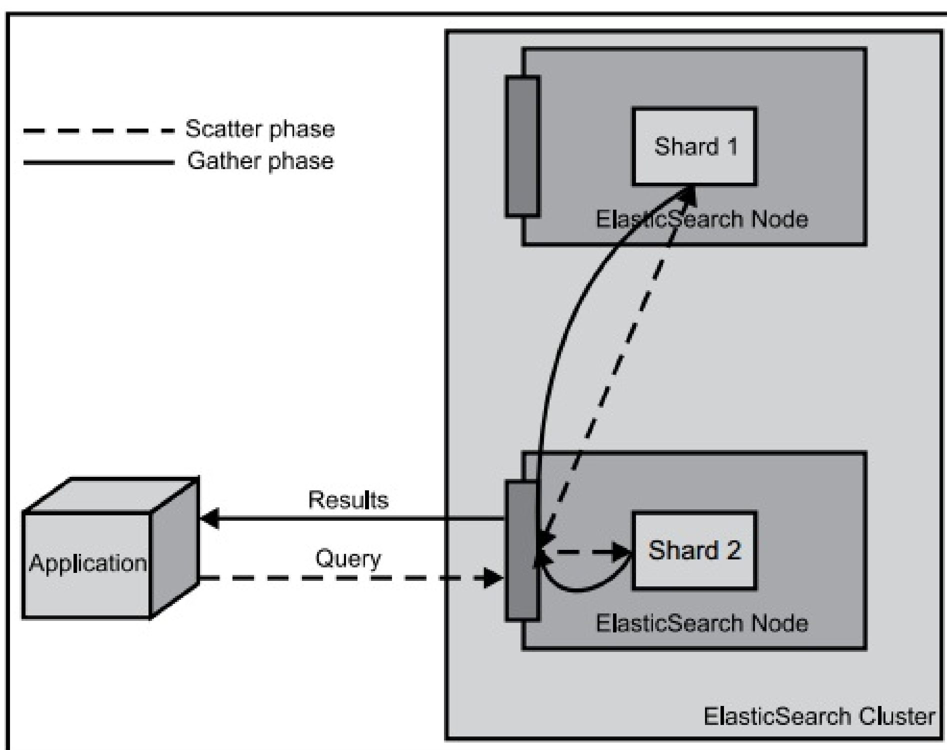


数据查询

查询API在ElasticSearch中有着很大的比重。通过使用Query DSL(基于JSON，用来构建复杂查询的语言)，我们能够：

- 使用各种类型的查询方式，包括：简单的关键词查询(termquery) ,短语(phrase)、区间(range)、布尔(boolean)、模糊(fuzzy)、跨度(span)、通配符(wildcard)、地理位置(spatial)等其它类型的查询方式。
- 通过组合简单查询构建出复杂的查询。
- 过滤文档，去除不符合标准的文档而且不影响打分排序。
- 查找给定文档的相似文档。
- 查找给定短语的搜索建议和查询短语修正。
- 通过faceting构建动态的导航和数据统计
- 使用prospective search而且找到匹配写定文档的查询语句。(关于*prospective search*，似乎是一种推送方式。即把用户的查询语句存储到索引中，如果新的文档添加到索引中，就把文档关联到匹配的查询语句中。这种查询适合于新闻、博客等会定时更新的应用场景)

关于数据查询，其核心点在于查询过程不是一个简单、单一的流程。通常这个过程分为两个阶段：查询分发阶段和结果汇总阶段。在查询分发阶段，会从各个分片中查询数据；在结果汇总阶段，会把从各个分片上查询到的结果进行合并、排序等其它处理过程，然后返回给用户。



用户可以通过指定搜索类型来控制查询的分发和汇总过程，目前搜索类型只有6种可选值。在Packt的出版的《ElasticSearch Server》一书中，已经讲述了查询范围(query scope)这一知识点.

索引参数设置

前面已经提到ElasticSearch索引参数的自动化配置和文档结构及域类型的自动识别。当然，ElasticSearch也允许用户自行修改默认配置。用户可以自行配置很多参数，比如通过mapping配置索引中的文档结构，设置分片(shard)和副本(replica)的的个数，设置文本分析组件……

集群管理和监控

通过管理和监控部分的API，用户可以更改集群的设置。比如调整节点发现机制(discovery mechanism) 或者更改索引的分片策略。用户可以查看集群状态信息，或者每个节点和索引和统计信息。集群监控的API非常广泛，相关的使用案例将会在 第*5*章 管理*ElasticSearch*。

本章小结

在本章我们学习了Apache Lucene的架构、工作原理、文本分析过程，以及如何使用Lucene的查询语言。此外，也探讨了ElasticSearch的基本概念、架构、内部通信。

下一章，读者将会了解Apache Lucene默认的打分公式、查询重写(query rewrite) 机制及它是如何工作的。此外，我们还将探讨ElasticSearch的一些功能，例如查询重排序、近实时搜索、批量搜索。我们也将探讨如何用update API更新部分文档，如何对数据进行排序、如何使用filtering机制提升搜索性能。最后，也会解析如何权衡faceting机制中filter和scope的使用。

第2章 强大的户查询语言DSL

在前面 的章节里，我们介绍了什么是Apache Lucene以及它的架构是怎样的，还有文件分析步骤的处理方式。此外，我们也明白了是Lucene查询语言是什么以及如何应用。我们也论述了ElasticSearch、它的架构和核心概念。在本章，我们将深入入探究ElasticSearch的Query DSL相关内容。在学习高级查询之前还是先了解一下Lucene的打分公式。通过本章内容的学习，我们将学习到：

- Apache Lucene的打分公式是如何工作的
- 查询重写机制是什么
- 查询的重排序是如何工作的
- 在一个请求中如何发送多个近实时数据获取命令
- 在一个请求中如何发送多条查询语句
- 结果集中有内嵌文档和多值域文档时如何进行排序
- 如何更新已经添加到索引中的文档
- 如何使用filter机制优化我们的查询
- 如何在ElasticSearch的faceting功能中使用filters和scopes

Lucene默认的打分算法

当谈论到查询的相关性，很重要的一件事就是对于给定的查询语句，如何计算文档得分。首先要弄清楚的是文档得分是什么。文档得分是一个用来描述查询语句和文档之间匹配程度的变量。在本节，我们将学习Lucene默认的打分机制：**TF/IDF(term frequency/inverse document freqecy)**算法，以及它是如何对相关文档进行打分排序。理解默认的打分算法对设计复杂查询语句来说至关重要，特别是在决定各个查询子句权重的时候。

匹配文档的打分因子

当一个文档出现在了搜索结果中，这就意味着该文档与用户给定的查询语句是相匹配的。Lucene会对匹配成功的文档给定一个分数。至少从Lucene这个层面，从打分公式的结果来看，分数值越高，代表文档相关性越高。 自然而然，我们可以得出：两个不同的查询语句对同一个文档的打分将会有所不同，但是比较这两个得分是没有意义的。用户需要记住的是：我们不仅要避免去比较不同查询语句对同一个文档的打分结果，还要避免比较不同查询语句对文档打分结果的最大值。这是因为文档的得分是多个因素共同影响的结果，不仅有权重(boosts)和查询语句的结构起作用，还有匹配关键词的个数，关键词所在的域，查询归一化因子中用到的匹配类型……。在极端情况下，只是因为我们用了自定义打分的查询对象或者由于倒排索引中词的动态变化，相似的查询表达式对于同一个文档都会产生截然不同的打分。

暂时还是先回来继续探讨打分机制。为了计算出一个文档的得分，我们必须考虑如下的因素：

- 文档权重**(Document boost)**：在索引时给某个文档设置的权重值。
- 域权重**(Field boost)**：在查询的时候给某个域设置的权重值。
- 调整因子**(Coord)**：基于文档中包含查询关键词个数计算出来的调整因子。一般而言，如果一个文档中相比其它的文档出现了更多的查询关键词，那么其值越大。
- 逆文档频率**(Inerse document frequency)**：基于Term的一个因子，存在的意义是告诉打分公式一个词的稀有程度。其值越低，词越稀有(这里的值是指单纯的频率，即多少个文档中出现了该词；而非指Lucene中idf的计算公式)。打分公式利用这个因子提升包含稀有词文档的权重。
- 长度归一化**(Length norm)**:基于域的一个归一化因子。其值由给定域中Term的个数决定(在索引文档的时候已经计算出来了，并且存储到了索引中)。域越的文本越长，因子的权重越低。这表明Lucene打分公式偏向于域包含Term少的文档。
- 词频**(Term frequency)**：基于Term的一个因子。用来描述给定Term在一个文档中出现的次数，词频越大，文档的得分越大。
- 查询归一化因子**(Query norm)**：基于查询语句的归一化因子。其值为查询语句中每一个查询词权重的平方和。查询归一化因子使得比较不同查询语句的得分变得可行，当然比较不同查询语句得分并不总是那么易于实现和可行的。

TF/IDF打分公式

接下来看看打分公式的庐山真面目。如果只是为了调整查询语句之间的关联关系，用户不必去理解它的原理。但是至少要知道它是如何工作的。

Lucene概念上的打分公式

TF/IDF公式的概念版是下面这个样子的：

$$score(q,d) = coord(q,d) * queryBoost(q) * \frac{V(q) * V(d)}{|V(q)|} * lengthNorm(d) * docBoost(d)$$

上面的公式展示了布尔信息检索模型和向量空间信息检索模型的组合。我们暂时不去讨论它，直接见识实际应用的公式，它是在Lucene实现并且正在使用的公式。



关于信息检索中的布尔模型和向量空间模型不在本书的知识范围。如果想了解更多的相关知识，可以从 http://en.wikipedia.org/wiki/Standard_Boolean_model 和 http://en.wikipedia.org/wiki/Vector_Space_Model 里面了解。

Lucene实际应用的打分公式

现在看看Lucene实际应用的打分公式长啥样：

$$score(q,d) = coord(q,d) * queryNorm(q) * \sum_{t\ in\ q} (tf(t\ in\ d) * idf(t)^2 * boost(t) * norm(t,d))$$

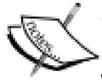
可以看到，文档的分数实际上是由查询语句q和文档d作为变量的一个函数值。打分公式中有两部分不直接依赖于查询词，它们是coord和queryNorm。 公式的值是这样计算的，coord和queryNorm两大部分直接乘以查询语句中每个查询词计算值的总和。

另一方面，这个总和也是由每个查询词的词频(tf)，逆文档频率(idf)，查询词的权重，还有norm，也就是前面说的length norm相乘而得的结果。

听上去有些复杂吧？不用担心，这些东西不需要全部记住。用户只需要知道在进行文档打分的时候，哪些因素是起决定作用的就可以了。基本上，从前面的公式中可以提炼出以下的几个规则：

- 匹配到的关键词越稀有，文档的得分就越高。
- 文档的域越小(包含比较少的Term)，文档的得分就越高。
- 设置的权重(索引和搜索时设置的都可以)越大，文档得分越高。

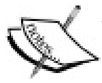
正如我们所看到的那样，Lucene会给具有这些特征的文档打最高分：文档内容能够匹配到较多的稀有的搜索关键词，文档的域包含较少的Term，并且域中的Term多是稀有的。



如果想了解更多关于Apache Lucene TF/IDF打分公式，请关注Apache Lucene Javadocs中的TFIDFSimilarity类，访问网址：http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html.

从ElasticSearch的角度看打分排序

最为重要的是利用Lucene构建起来的ElasticSearch允许用户修改默认的打分算法(了解更多相关的知识请参考第3章 索引底层控制 中 修改*Lucene*打分算法一节的内容)。但是要记住，ElasticSearch不仅仅是Lucene简单的封装，因为在ElasticSearh中，文档排序并非完全依赖于Apache Lucene的打分算法。ElasticSearch中实现了多种不同的查询类型，这些查询类型可以完全控制文档打分的计算方式(比如 custom_boost_facotr query,constant_score query,custom_score query)，ElasticSearch允许通过脚本定制文档的打分方式。用户可以利用ElasticSearch 0.90版本支持的重排序(rescore)机制，重新计算搜索到的文档。也可以通过其它的查询方式处理topN 结果集，不一而足。



如果想了解更多关于Apache Lucene 的query类型，请参考相关的 Javadocs。比如：http://lucene.apache.org/core/4_5_0/queries/org/apache/lucene/queries/package-summary.html.

查询重写机制

如果你曾经使用过很多不同的查询类型，比如前缀查询和通配符查询，从本质上上，任何的查询都可以视为对多个关键词的查询。可能用户听说过查询重写(query rewrite)，ElasticSearch(实际上是Apache Lucene很明显地)对用户的查询进行了重写，这样做是为了保证性能。整个重写过程是把从Lucene角度认为原始的、开销大的查询对象转变成一系列开销小的查询对象的一个过程。

以前缀查询为例

展示查询重写内部运作机制的最好方法是通过一个例子，查看例子中用户输入的原查询语句中的term在内部被什么Term所取代。假定我们的索引中有如下的数据：

```
    curl -XPUT 'localhost:9200/clients/client/1' -d
'{
  "id":"1","name":"Joe"
}'
curl -XPUT 'localhost:9200/clients/client/2' -d
'{
  "id":"2","name":"Jane"
}'
curl -XPUT 'localhost:9200/clients/client/3' -d
'{
  "id":"3","name":"Jack"
}'
curl -XPUT 'localhost:9200/clients/client/4' -d
'{
  "id":"4","name":"Rob"
}'
curl -XPUT 'localhost:9200/clients/client/5' -d
'{
  "id":"5","name":"Jannet"
}'
```



源码下载
所有购买了Packt出版的图书的读者都可以用自己的账户从http://www.packtpub.com. 下载源代码文件。如果读者通过其它的途径下载本书，则需要通过http://www.packtpub.com/support 来注册账号，然后网站会把

源码通过e-mail发送到你的邮箱。

我们的目的是找到所有以字符 j 开头的文档。这个需求非常简单，在 client索引上运行如下的查询表达式：

```
curl -XGET 'localhost:9200/clients/_search?pretty' -d '{
    "query" : {
        "prefix" : {
            "name" : "j",
    "rewrite" : "constant_score_boolean"
        }
    }
}'
```

我们用的是一个简单的前缀查询；前面说过我们的目的是找到符合以下条件的文档：name域中包含以字符 j 开头的Term。我们用rewrite参数来指定重写查询的方法，关于该参数的取值我们稍后讨论。运行前面的查询命令，我们得到的结果如下：

```
{ ...
"hits" : {
  "total" : 4,
  "max_score" : 1.0,
  "hits" : [ {
    "_index" : "clients",
    "_type" : "client",
    "_id" : "5",
    "_score" : 1.0, "_source" : {"id":"5","name":"Jannet"}
  }, {
    "_index" : "clients",
    "_type" : "client",
    "_id" : "1",
    "_score" : 1.0, "_source" : {"id":"1", "name":"Joe"}
  }, {
    "_index" : "clients",
    "_type" : "client",
    "_id" : "2",
    "_score" : 1.0, "_source" : {"id":"2", "name":"Jane"}
  }, {
    "_index" : "clients",
    "_type" : "client",
    "_id" : "3",
    "_score" : 1.0, "_source" : {"id":"3", "name":"Jack"}
  } ]
} }
```

重排序

批处理

查询结果的排序

update API

使用filters优化查询

filters和scope在ElasticSearch Faceting模块的应用

本章小结

第3章 索引底层控制

第4章 探究分布式索引架构

在前面的章节里，我们已经学习了如何使用不同的打分公式，也了解了使用这些打分公式的好处。我们也学习了何如使用不同的倒排表结构来改变索引数据的方式。此外，我们也学习了自如应用近实时搜索和数据实时获取(real-time GET),了解了检索器(searcher)重启(reopen)背后的意义。我们也探讨了多语言数据的处理，也学习了配置事务日志来实现业务需求。最后，我们学习段合并(segments merging)、合并策略和合并任务执行计划相关的知识。在本章，读者将了解如下的知识：

- 如何为集群选择合适的分片(shard)和分片副本(replica)的数量
- 路由是什么，路由对ElasticSearch起着什么样的作用
- ShardAllocator是如何工作的，如何配置它
- 如何根据业务需求来调整分片分发机制
- 如何选择合适的分片来执行相关的命令
- 如何结合现有的知识配置一个真实业务场景的群集
- 数据和搜索并发量增长时如何应对

选择恰当的分片数量和分片副本数量

最开始使用ElasticSearch时，一般都是创建一个索引，导入数据，然后发送查询命令检索数据。我们确信系统运行庚子，至少在最开始，数据量不大而且QPS(Query Per Second)也不高的时候运行良好。在幕后，ElasticSearch创建了一些分片来存储数据，也可能还会创建分片副本(例如，如果用默认配置)，而且用户在配置方面也不用过多地操心。

当应用程序规模增长起来，越来越多的数据需要进行索引，QPS也变得越来越高。这个时候，量变就引起质变了。慢慢地问题也就出现了(读者可以阅读 知识的灵活运用一节的内容来了解应用程序规模增长的对应方法) 。这时候，就该思想如何规划索引数据、优化配置使得应用程序的处理能力与规模增长相同步。在本章，作者将给出一些应对相关问题的指导方针。然而对应问题并没有一个统一的标准，不同的应用场景有着不同的特点和需求，这些特点和需求不仅体现在索引结构上，同时也体现在配置上面。例如：整个索引的数据规模、查询类型、预期的吞吐量都是不尽相同的。

分片和过度分配(over allocation)

关于索引分片，在第一章 认识*ElasticSearch*里面已经有介绍，这里简单回顾一下。索引分片就是把索引数据切分成多个小的索引块，这些小的索引块能够分发到同一个集群中的不同节点。在检索时，检索的结果是该索引每个分片上检索结果的总和(尽管在某些场景中“总和”并不成立：单个分片有可能存储了所有的目标数据)。默认情况下，ElasticSearch会为每个索引创建5个主分片，就算是单结点集群亦是如此。像这样的冗余就称为过度分配：这种场景下，这种分配方式似乎是多此一举，而且只会在索引数据(把文档分发到多个分片上)和检索(必须从多个分片上查询数据然后全并结果)的时候增加复杂度，乐享其成的是，这些复杂的事情是自动处理的，但是为什么ElasticSearch要这样做呢？

假定我们有一个构建在单个分片上的索引。这意味着如果我们的应用程序规模增长到单机无法处理的情况下时，问题就来了。当索引中有数据时，当前版本的ElasticSearch是无法将分片中的数据切分成多个小块的：我们只能在创建索引的时候指定好分片的数量。唯一的解决办法就是重新创建一份多个分片的索引，然后将原来的数据重新索引(动词)到新的索引(名词)中。然而这样的处理方案需要时间和服务器资源，比如CUP时间、内存、大容量存储器。而且有可能我们并没有时间和上面提到的那些资源。换个角度，通过使用过度分配策略，我们可以在必要的时候添加一台新的安装了ElasticSearch的服务器。当新的节点添加进来时，ElasticSearch会自动对集群进行负载均衡，在不需要重新索引数据的前提下将部分索引数据转移到新的机器上。ElasticSearch作者设置的默认配置(5个主分片和一个分片副本)是权衡了数据增长的可能性和合并不同分片数据的最终开之后的最终选择。

默认配置适用于大部分场合。那么问题来了：当我们在什么时个需要增加或者减少分片数量，让分片数量尽可能少一些？

第一个问题的答案显而易见。如果数据集的大小有限制而且严格定义好的，可以只使用一个分片。如果不是的，大拇指法则表明最佳的分片数量取决于节点数量。所以，如果计划将会到10个节点，那么就需要配置10个分片。需要记住的重点是：考虑到高可用性和吞吐量，分片副本也是需要声配置的。分片副本像普通分片一样也占用空间。如果为每个分片指定一个拷贝(number_of_shards =1),那么就需要20个分片 ：10个主分片和10个分片副本。这个简单的公式可以总结如下：

*Max number of nodes = Number of shards * (number of replicas +1)*

换句话说，如果你计划用10个分片和2个分片副本，那么最大的节点数是30。

一个关于过度分配的正例

如果读者仔细阅读了本章前面部分的内容，应该确信配置最小数量的分片是明智的选择。但是有时分配更多的分片却比较方便，因为每个分片都是独立的Lucene索引，更多的分片意味着在单个较小的索引上进行操作(特别是数据索引操作)会比较高效。对于有的应用场景来说，上文是选择多分片不错的理由。当然多分片同时也带了额外的开销：分发搜索命令到每个分片以及分片结果的合并。这个开销在某些特定的应用场景是可以避免的，这类应用场景的一个特点是：查询命令总是会被具体的参数过滤。像多租户系统，每个查询命令都被限定在确定用户上下文中。解决问题的思想很简单：把每个用户的数据都存储在一个分片上，而且只在查询的时候使用该分片。这也是ElasticSearch的路由功能大显身手的地方(我们将在本章的 路由功能浅谈 一节详细讨论这一知识点)。

多分片 vs 多索引

也许读者会感到疑惑，如果说分片实际上是一段小的Lucene索引，那么真实的ElasticSearch索引是什么呢？它们之间的区别是什么呢？从技术上来说，它们是一样的，但是会有一个额外的特性只能工作在多索引 或者 多分片结构上。对于数据分片，可以通过路由功能或者在指定的分片上执行查询命令功能实现将查询命令定位到特殊的分片上去。对于多索引结构，更通用的机制在于多索引地址的处理上，查询语句可以通过/index1,index2.../符号联合多个索引进行查询。在查询时，用户也可以通用使用别名(aliasing)特性将多个索引以一个索引的方式呈现给用户，索引分片时也可以用别名这个功能。在负载均衡处理逻辑上，容易发现两者之间更多的不同点，然而自动化程度较低的多索引可以在索引数据时，以通过人工操作强行将多个索引部署在一个节点上的方式部分地隐藏这一点。

分片副本

索引分片机制用来存储超过单个节点存储容量的数据，分片副本用来应对不断攀升的吞吐量以及确保数据的安全性。当一个节点的主分片丢失，ElasticSearch可以把任意一个可用的分片副本推举为主分片。在默认情况下，ElasticSearch会创建一个分片副本。然而分片副本的数量可以通过设置相关的API随时更新，这一点与主分片是不同的。分片副本的动态更新

功能使得创建应用程序时十分方便，查询吞吐量可以随着分片副本数量的增加而增长，与此同时，使用分片副本还可以处理查询的发并量。使用分片副本的缺点也是显而易见的：额外的存储空间开销，从主分片复制数据到分片副本时的开销。选定分片副本的数量时，还需要考虑到现阶段需要多少副本。如果分片副本的数量太多，那么就会浪费存储空间和ElasticSearch的资源，实际上一些分片副本并没有起作用。相反地，如果没有设置分片副本，如果主分片出现了故障，数据就会丢失。

路由功能浅谈

在本章 选择恰当的分片数量和分片副本数量 一节中，已经提到使用路由功能可以只在一个分片上执行查询命令，作为提高系统吞吐量的一种解决方案。接下来作者将详细地介绍这一功能。

分片和分片中数据

通常情况下，ElasticSearch是如何把数据分发到各个分片中，哪个分片存储哪一类的文档等细节并不重要。因为查询时，将查询命令分发到每个分片就OK了。唯一的关键点在于算法，将数据均等地分配到各个分片的算法。在删除或者更新文档时，情况就会变得有点复杂了。实际上，这也不是什么大问题。只要保证分片算法在处理文档时，对于相同的文档标识生成相同的映射值就可以了。如果我们有这样的分片算法，ElasticSearch就知道在处理文档时，如何定位到正确的分片。但是，在选择文档的存储分片时，采用一个更加智能的办法不就更省事儿了吗？比如，把某一特定类型的书籍存储到特定的分片上去，这样在搜索这一类书籍的时候就可以避免搜索其它的分片，也就避免了多个分片搜索结果的合并。这就是路由功能(routing)的用武之地。路由功能向ElasticSearch提供一种信息来决定哪些分片用于存储和查询。同一个路由值将映射到同一个分片。这基本上就是在说："通过使用用户提供的路由值，就可以做到定向存储，定向搜索。"

路由功能的简单应用

我们将通过一个例子来说明ElasticSearch是如何分配分片，哪些文档会存储在特定的分片上这一过程，为了使细节更清楚，我们借助一个第三方插件。这个插件将帮助我们更生动形象地了解ElasticSearch处理数据的过程。用如下的命令来安装插件：

```
bin/plugin -install karmi/elasticsearch-paramedic
```

重启ElasticSearch后，用浏览器打开http://localhost:9200/_plugin/paramedic/index.html 我们将能看到显示索引的各种信息的统计结果的页面。例如：最值得关注的信息是集群的状态和每个索引下分片及分片副本的相关信息。

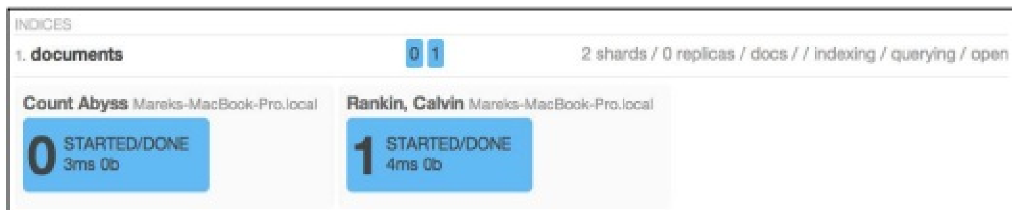
启动两个节点的ElasticSearch集群，用如下的命令创建索引：

```
curl -XPUT localhost:9200/documents -d '{
  settings: {
    number\_of\_replicas: 0,
    number\_of\_shards: 2
  }
}'
```

我们创建了具有2个分片0个分片副本的索引。这意味着该集群最多有两个结点，多余的节点无法存储数据，除非我们增加分片副本的数量(读者可以回顾本章选择恰当的分片数量和分片副本数量一节内容了解相关知识)。创建索引后，下一步的操作就是添加数据了。我们用如下的命令来进行数据添加：

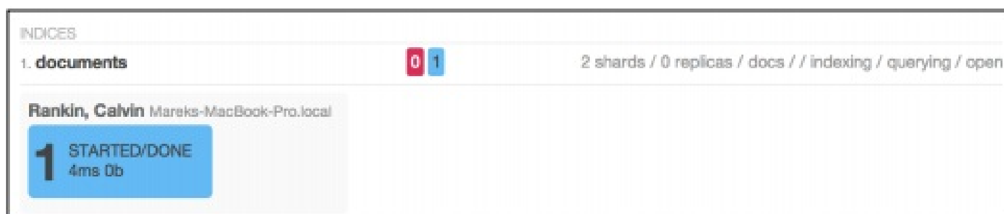
```
curl -XPUT  localhost:9200/documents/doc/1 -d '{ "title" : "Document No. 1" }'
curl -XPUT  localhost:9200/documents/doc/2 -d '{ "title" : "Document No. 2" }'
curl -XPUT  localhost:9200/documents/doc/3 -d '{ "title" : "Document No. 3" }'
curl -XPUT  localhost:9200/documents/doc/3 -d '{ "title" : "Document No. 4" }'
```

添加数据完成后，Paramedic插件显示集群中存在两个分片，截图如下：



在节点所呈现的信息中，我们一样能够找到值得关注的信息。集群中的每个节点都存储着两个文档，这让我们容易得出如下的结论：分片算法完美地完成了数据分片的任务；集群中有一个由两个分片构建成的索引，索引对每个分片中文档的数量进行了均等分配。

现在我们做一些破坏，关闭一个节点。用Paramedic插件，我们将看到类似下图的截图：



我们关注的第一条信息是集群的状态已经变成了红色状态。这意味着至少有一个主分片已经丢失，这就表明有一部分数据已经失效，同时有一部分索引也失效了。尽管如此，在ElasticSearch集群上仍然可以进行查询操作。ElasticSearch集群把决定权交给了开发者：决定返回给用户不完整的数据还是阻塞用户查询。先看看查询命令的返回结果吧：

```
{
  "took" : 30,
  "timed\_out" : false,
  "\_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 1,
    "failures" : [ {
      "index" : "documents",
      "shard" : 1,
      "status" : 500,
      "reason" : "No active shards"
    }]
  },
"hits" : {
  "total" : 2,
  "max\_score" : 1.0,
  "hits" : [ {
    "\_index" : "documents",
    "\_type" : "doc",
    "\_id" : "1",
    "\_score" : 1.0, "\_source" : { "title" : "Document No. 1" }
  }, {
    "\_index" : "documents",
    "\_type" : "doc",
    "\_id" : "3",
    "\_score" : 1.0, "\_source" : { "title" : "Document No. 3" }
  } ]
} }
```

正如你所看见的那样，ElasticSearch返回了分片失效的信息。我们看到分片1失效。在返回的结果集中，我们只看到了ID为1和3的文档。至少在主分片1重新连接到集群之前，其它的文档丢失。如果重新启动第二个节点，经过一段时间(取决于网络情况和gateway模块的参数设置)，集群将返回绿色状态，而且整个索引中的文档都可用。接下来，我们将用路由功能(routing)来做与上面一样的事情，并观察两者在集群中的不同点。

启用路由功能(routing)索引数据

通过路由功能，用户能够控制ElasticSearch用哪个分片来存储文档。路由的参数值是无关紧要的，可以由用户随意给出。关键点在于相同的参数值应该用来把不同的文档导向同一个分片。

将路由参数信息写入到ElasticSearch中有多种方式，最简单的一种是在索引文档的时候提供一个routingURL 参数。比如：

```
curl -XPUT  localhost:9200/documents/doc/1?routing=A -d '{ "title":"Document" }'
```

另外一种方式就是在创建文档时把_routing域放进去：

```
curl -XPUT  localhost:9200/documents/doc/1 -d '{ "title":"Document","\_routing":"A" }'
```

然而这种方式只有在mapping中定义了_routing域才会生效。例如：

```
"mappings": {
  "doc": {
    "\_routing": {
      "required": true,
      "path": "\_routing"

    }, "properties": {
      "title" : {"type": "string" }
    }
  }
}
```

让我们在这里停留一下。在本例中，我们用到了_routing域。值得一提的是path参数可以指向文档中任何not-analyzed域。这是一个非常强大的特性。例举routing特性的一个主要的优点：比如我们在文档中定义了library_id域，用来表示书籍所在的藏书室编号。当我们基于library_id域来实现路由功能，基于该藏书室检索相关图书将使检索过程更高效，这样做也是合乎逻辑的。

现在，我们回过头来继续讨论routing值的定义方式。最后一个办法是使用批处理索引。在下面的例子中，routing被放置到每个文档的头部信息块中。例如：

```
curl -XPUT localhost:9200/\_bulk --data-binary  '{
  "index" : {
    "\_index" : "documents", "\_type" : "doc", "\_routing" : "A"
  }
} { "title" : "Document No. 1"}'
```

既然已经了解routing是如何工作的，那么就回到我们例子中来。

启用路由功能(routing)索引数据

现在，我们对照前面的例子，依样画葫芦，除了一点不一样，那就是使用路由功能(routing)。第一件事就是删除所有旧的文档。如果不清空索引数据，在添加id相同的文档到索引中时，routing可能会把相同的文档写入到其它的分片上(经验证：索引中会存在id一样的两个文档，只不过在不同的分片中)。因此，运行如下的命令清空索引数据：

```
curl -XDELETE localhost:9200/documents/\_query?q=*:*
```

经过这一步，我们重新索引数据。但是这次要添加上routing信息。因此索引数据的命令如下：

```
curl -XPUT localhost:9200/documents/doc/1?routing=A  -d '{ "title" : "Document No. 1" }'
curl -XPUT localhost:9200/documents/doc/2?routing=B  -d '{ "title" : "Document No. 2" }'
curl -XPUT localhost:9200/documents/doc/3?routing=A  -d '{ "title" : "Document No. 3" }'
curl -XPUT localhost:9200/documents/doc/4?routing=A  -d '{ "title" : "Document No. 4" }'
```

routing参数指挥ElasticSearch把携带该参数的文档放到特定的分片中。当然这也并不意味着routing参数值不同的文档会被放置到不同的分片中。但是在我们的例子中，文档数目不多，routing参数不同，文档所在的分片也会不同。读者可以在Paramedic页面上验证：两个节点中一个节点上只有一个文档(它的routing值是B)，另一个节点中有3个文档(它们的routing值是A)。如果我们关闭一个节点，Paramedic页面上集群的状态将会变成红色状态，检索所有的文档，其结果如下：

```
{
  "took" : 1,
  "timed\_out" : false,
  "\_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 1,
      "failures" : [ { "index" : "documents", "shard" : 1, "status" : 500, "reason" :
      "No active shards"
    } ]
  }, "hits" : {
    "total" : 3,
    "max\_score" : 1.0,
    "hits" : [ {
      "\_index" : "documents",
      "\_type" : "doc",
      "\_id" : "1",
      "\_score" : 1.0, "\_source" : { "title" : "Document No. 1" }
    }, {
      "\_index" : "documents",
      "\_type" : "doc",
      "\_id" : "3",
      "\_score" : 1.0, "\_source" : { "title" : "Document No. 3" }
    }, {
      "\_index" : "documents",
      "\_type" : "doc",
      "\_id" : "4",
      "\_score" : 1.0, "\_source" : { "title" : "Document No. 4" }
    } ]
  }
}
```

在本例中，id为2的文档丢失了。集群中文档routing值为B的节点失效。如果运气不好，关闭了另一个节点，我们将丢失3个文档的数据。

routing功能用于检索

当我们能够很好的掌控routing机制时，routing能让我们在的数据检索变得更高效。如果我们只想从整个索引中获取特定的部分数据，为什么还要去一一检索所有的节点呢？对于那些带routing参数值为A的索引数据，我们只需简单地执行如下的查询命令，就可以检索到它

们。

> curl -XGET 'localhost:9200/documents/_search?pretty&q=*:*&routing=A'

我们仅仅是在查询命令中添加了一个routing参数，和一个我们感兴趣的参数值。对于上面的命令，ElasticSearch将返回如下的结果：

```
{ "took" : 1,
"timed\_out" : false,
"\_shards" : {
  "total" : 1,
  "successful" : 1,
  "failed" : 0,
  },
"hits" : {
  "total" : 3,
  "max\_score" : 1.0,
  "hits" : [ {
    "\_index" : "documents",
    "\_type" : "doc",
    "\_id" : "1",
    "\_score" : 1.0, "\_source" : { "title" : "Document No. 1" }
  }, {
    "\_index" : "documents",
    "\_type" : "doc",
    "\_id" : "3",
    "\_score" : 1.0, "\_source" : { "title" : "Document No. 3" }
  }, {
    "\_index" : "documents",
    "\_type" : "doc",
    "\_id" : "4",
    "\_score" : 1.0, "\_source" : { "title" : "Document No. 4" }
  } ]
} }
```

一切就像魔法一样。但是要注意，我们忘记去启动另一个节点，该节点上的分片中存储着用routing值为B索引的文档。尽管我们并没有索引的全景图，但是ElasticSearch的回复值并没有包含分片失效的信息。这证明带routing参数的查询命令只会到指定的分片上获取数据，而忽略其它的分片。如果用同样的命令，带上参数routing=B，我们将得到类似如下的异常信息：

> { "error" : "SearchPhaseExecutionException[Failed to execute phase [query_fetch], total failure; shardFailures {[_na_][documents][1]: No active shards}]", "status" : 500 }

对于集群的性能优化，routing机制是一个功能非常强大工具。它让我们能够根据应用程序的数据分类逻辑将文档分发到各个分片，通过它不仅节约服务器资源，还能构建出更加快捷的查询服务。

我们还要复述一个要点：Routing能够确保在索引阶段把routing值相同的文档分发到相同的分片上。但是，同一个分片可能会有多个不同的routing值。Routing允许我们在查询的时候限定查询的节点数量，但是无法取代过滤功能(filtering)的地位。这意味着查询命令无论是否启用routing功能，都应该带有相同的过滤器(filters)。

别名功能

最后，有一个能够简化routing功能的特性值得一提。如果读者是一个搜索引擎专家，那么在开发中，多半会选择对外隐藏搜索引擎的配置细节，以实现架构上的解耦，同时程序员在使用中不必关注搜索引擎的各种细节，以提升开发效率。对于程序员来说，一个理想的搜索引擎就是不必关注routing、分片、分片副本。通过别名功能，我们可以像使用普通的索引一样使用routing功能。例如：通过如下的命令可以创建一个别名：

```
curl -XPOST 'http://localhost:9200/\_aliases' -d ' {
  "actions" : [
    {
      "add" : {
        "index" : "documents",
        "alias" : "documentsA",
        "routing" : "A"
      }
    }
  ]
}'
```

在前面的例子中，我们创建了一个虚拟的索引，命名为documentsA，虚拟索引中的数据都来自于documents 索引。更重要的是：搜索的分片将被限制在routing值为A的分片中。正因为有这个特性，只需要把别名documentsA提供给开发人员，他们就能像使用普通的索引一样进行查询和索引的操作。从而屏蔽了相关的细节。

多个**routing**值的联合

ElasticSearch允许在单个查询请求中指定多个routing值。多个routing值也就意味着会在多个分片上搜索，搜索哪个分片取决于给定routing值的文档会被映射到哪个分片。下面的查询命令就是一个简单的例子:

> curl -XGET 'localhost:9200/documents/_search?routing=A,B'

执行查询命令后，ElasticSearch会把搜索请求发送到索引中的所有分片。因为routing值为A代表着索引的一个分片，routing值为B代表着索引的第二个分片，而索引一共只有2个分片。

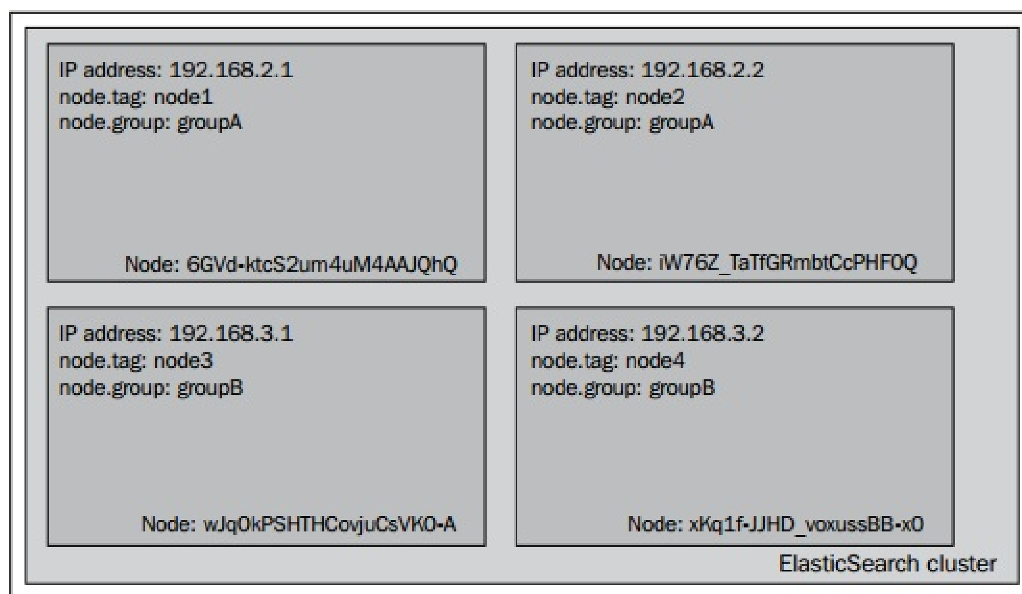
当前，别名也是支持多个routing值的。下面的例子演示了这个特性。

```
curl -XPOST 'http://localhost:9200/\_aliases' -d ' {
  "actions" : [
    {
      "add" : {
        "index" : "documents",
        "alias" : "documentsA",
        "search\_routing" : "A,B",
        "index\_routing" : "A"
      }
    }
  ]
}'
```

上面的例子中用到了两个我们没有提到的配置参数，对于搜索和索引两个过程，我们可以配置不同的值。上例中，我们设定在查询时(search_routing参数)，两个routing参数值(A和B)将被用到；在索引时(index_routing参数)，只有一个routing值(A)会被用到。提示一下，索引过程是不支持多个routing参数，要记得选择合适的过滤器(在别名中也可以配置该参数)。

调整集群的分片分配

在*ElasticSearch Server*一书中，我们探讨了如何强制改变分片的分配方式，如何取消、如何使用一条API命令在集群中转移分片。然而在谈论到分片分配时，ElasticSearch允许我们做的不止如此，我们还可以定义以系列用于分片分配的规则。例如，假定一个4-节点的集群，图示如下：



正如你所看到的一样，集群由4个节点构成。每个节点都绑定了一个特定的IP地址，同时每个节点也拥有tag属性和group属性(可以在elasticsearch.yml文件中设置node.tag和node.group属性)。集群用来展示分片分配过滤器是如何工作的。group属性和tag属性可以用任意其它的名字替换，只需要把node作为自定义属性的前缀即可。比如你喜欢用属性名，party，就只需要把node.party:party1添加到你的elasticsearch.yml文件中即可。

Allocation awareness 配置

Allocation awareness制允许用户使用泛型参数来配置分片及分片副本的分配。为了演示allocation awareness的工作方式，我们使用我们的样例集群。为了集群的演示效果，我们在elasticsearch.yml文件中添加如下的属性：

```
cluster.routing.allocation.awareness.attributes:group
```

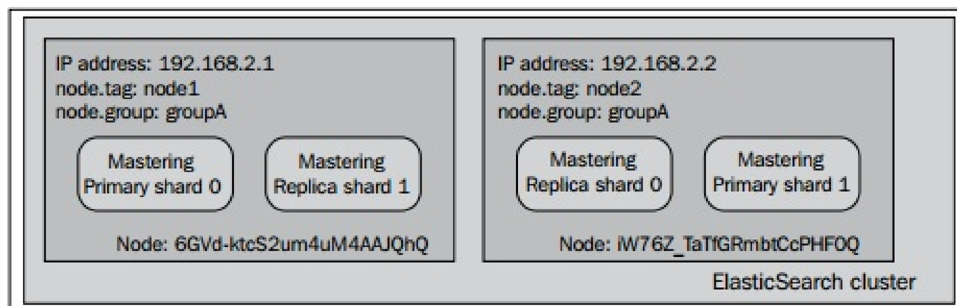
这条配置命令用来通知Elasticsearh使用node.group属性作为集群的awareness参数。

> 设置cluster.routing.allocation.awareness.attributes属性的参数时，可以指定多个值。比如:
>
> ```
> cluster.routing.allocation.awareness.attributes:group,node
> ```

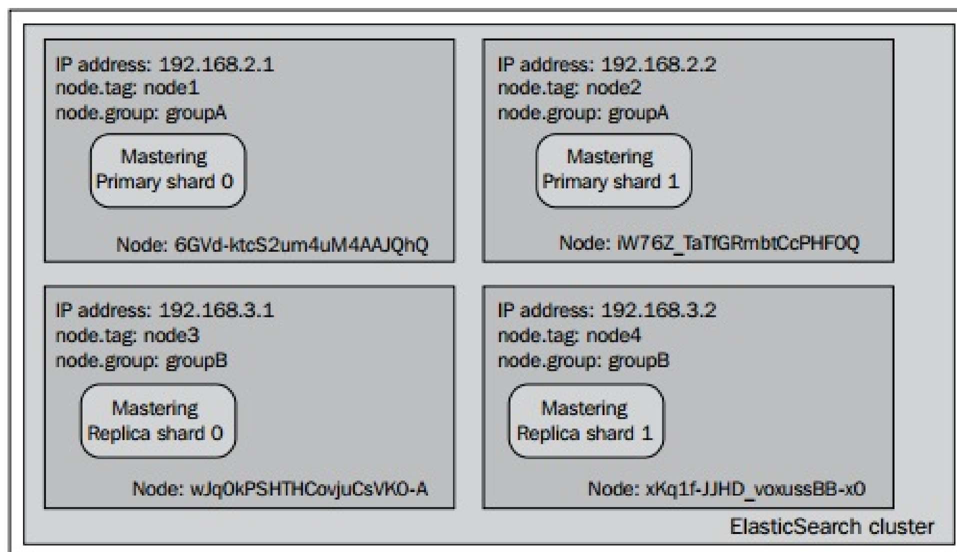
参数设置好以后，我们先启动两个节点，两个节点的node.group值都是groupA，并且用如下的命令创建索引：

```
curl -XPOST 'localhost:9200/mastering' -d '{
  "settings" : { "index" : {
    "number\_of\_shards" : 2,
    "number\_of\_replicas" : 1
  } }
}'
```

这个命令执行后，我们的2-节点集群看起来或多或少地类似于下面的图形：



正如所看见的那样，索引的分片平均分配到了两个节点中。现在，我们看看当启动剩下的两个节点时(node.group属性值设置为groupB)将会发生什么？



注意两者的不同点：主分片并没有从原来分配的节点中移出，反而是分片副本移动到了node.grooup值不同的节点中，这正是我们所希望的结果。在集群中使用了shard allocation awareness功能后，ElasticSearch不会把决定allocation awareness的属性(在本例中是node.group值)相同的分片或者分片副本分配到同一个节点中。该功能典型的用例是把集群拓扑结构部署到物理机或者虚拟机时，确保你的集群不会出现单点故障问题。

> 请记住在使用allocation awareness功能时，分片不会被分配到没有设置相应属性的节点上。所在在我们的案例中，分片分配机制不会考虑分配分片到没有设置node.group属性的节点。

Forcing allocation awareness

(文本的描述不并清晰，参考

http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/modules-cluster.html，一看就懂)

当我们事先知道awareness属性的取值范围并且不希望集群中有过多的分片副本时，使用forcing allocation awareness机制会很方便。比如，不希望集群中负载了过多的分片副本，我们可以强制allocation awareness只在有确定参数值时起作用。我们可以指定cluster.routing.allocation.awareness.force.zone.values属性的值，这是一个多值属性，多个值可以用逗号区分开来。比如，如果我们希望allocation awareness只在node.group属性的值为groupA和groupB生效时，我们可以在elasticsearch.yml文件中加入如下的代码：

```
cluster.routing.allocation.awareness.attributes: group
cluster.routing.allocation.awareness.force.zone.values: groupA, groupB
```

过滤

ElasticSearch允许用户从整个集群或者索引的层面上配置allocation机制。在集群层面上配置allocation机制时，我们可以用如下的属性前缀：

- cluster.routing.allocation.include
- cluster.routing.allocation.require
- cluster.routing.allocation.exclude

如果是在索引层面的分配，我们用如下的属性前缀：

- index.routing.allocation.include
- index.routing.allocation.require
- index.routing.allocation.exclude

上面提到的前缀可以和elasticsearch.yml文件中定义的属性(tag属性和group属性)结合起来，而且还有一个命名为_ip的属性允许用户匹配或者排除一些特定IP的节点。比如：

```
cluster.routing.allocation.include.\_ip:192.168.2.1
```

如果我们希望把group属性值为groupA的节点包括进来，我们可以设置如下的属性：

```
cluster.routing.allocation.include.group:groupA
```

注意我们使用cluster.routing.allocation.include属性的方式是以它为前缀并和其它属性的名字串联起来，在本例中是group属性。

include,exclude,required属性的意义

如果读者仔细观察了前面提到的参数，应该能注意到它们分为三种：

- include:这种类型将导致所有定义了该参数的节点都会被包括进来。如果配置多种include的条件，那么在进行分片分配的时候，只要有一个条件满足，节点就会被allocation考虑进去。比如，如果我们在配置的cluster.routing.allocation.include.tag参数中中添加2个值：node1和node2，那么最终索引的分片会分配到第一个节点和第二个节点中(从左到右数)。总结一下：对于带有include allocation参数类型的结点，

ElasticSearch会考虑把分片分配到该节点，但是并不意味着ElasticSearch一定会把分片分配到节点。

- require:这个属性是ElasticSearch 0.90版本引入到allocation filter中去的。它需要节点的所有相关属性值都满足它设定的值。比如，如果我们往配置文件中添加cluster.routing.allocation.require.tag参数并设其值为node1，添加cluster.routing.allocation.require.group参数并设其值为groupA，最终所有的分片将会分配到第一个节点(IP值为192.168.2.1的节点)
- exclude:这个属性允许我们在allocation过程中排除匹配属性值的节点。比如，如果我们设置cluster.routing.allocation.include.tag的值为groupA，最终我们的索引分片只会分配到IP值为192.168.3.1和192.168.3.2的节点上(例子中的第3和第4个节点)。



属性值可以使用简单的正则表达式。比如，如果我们包含所有group属性中属性值以字符串group开头的结点，可以设置cluster.routing.allocation.include.group的值为group*。在我们的样例集群中，它会匹配到group参数值为groupA和groupB的节点。

运行时**allocation**参数更新

除了可以在elasticsearch.yml文件中设置前面讨论的属性，当集群在线时，我们也可以通过update API来实时更新这些参数。

索引层面的参数更新

如果想更新给定索引(比如例子中的mastering索引)的配置信息，我们就要运行运行如下的命令：

```
curl -XPUT 'localhost:9200/mastering/\_settings' -d '{
"index.routing.allocation.require.group": "groupA" }'
```

正如你所看到的，命令被发送到给定索引的_settings端点。在一条命令中可以包含多个属性。

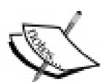
集群层面的参数更新

如果想更新整个集群的配置信息，我们就要运行运行如下的命令：

```
curl -XPUT 'localhost:9200/\_cluster/\_settings' -d '{ "transient" : {
"cluster.routing.allocation.require.group": "groupA" } }'
```

正如你所看到的，命令被发送到_cluster_settings端点。在一条命令中可以包含多个属性。请记住上面命令中transient关键字，它表示设置的属性在集群重启后就不再生效。如果希望设置的属性永久生效，用persistent属性代替transient属性就可以了。下面的命令示例将会使用用户设置在系统重启后依然生效：

```
curl -XPUT 'localhost:9200/\_cluster/\_settings' -d '{ "peristent" : {
"cluster.routing.allocation.require.group": "groupA" } }'
```



请注意，在相应的结节上运行上面的命令时，会导致分片在节点间的移动。

限定每个分片上节点的数量

除了前面提到的那些属性，我们也允许用户自定义每个节点上能够分配的分片(主分片和分片副本)数量。为了实现这个功能，用户需要在index.routing.allocation.total_shards_per_node属性中设置相应的值。比如在elasticsearch.yml文件中，我们应该设置如下：

```
index.routing.allocation.total\_shards\_per\_node:4
```

这个属性规定了每个节点中，单个索引最多允许分配4个分片。这个属性也可以通过update API在线上实时修改：

```
curl -XPUT  'localhost:9200/mastering/\_settings'  -d  '{
"index.routing.allocation.total\_shards\_per\_node": "4" }'
```

现在，让我们看看在elasticsearch.yml文件中配置了allocation的相关属性后，几个单索引集群会变成什么样。

"结点包含"属性

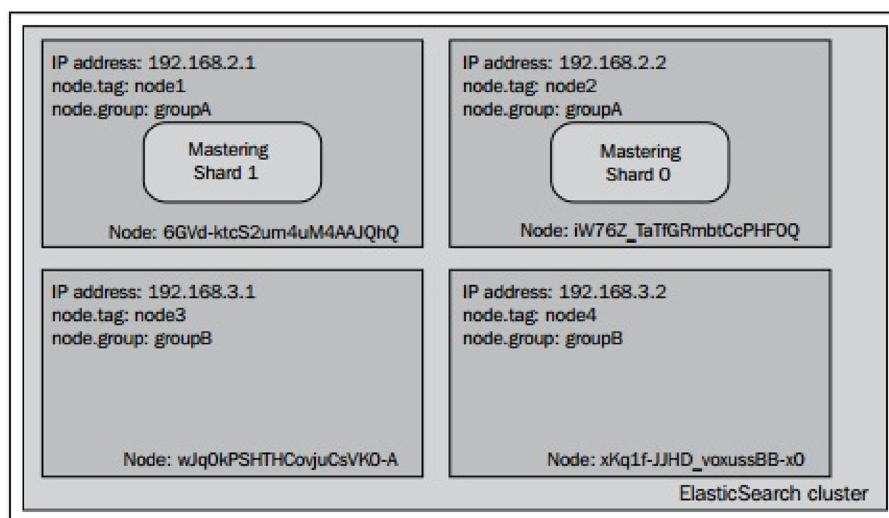
现在通过我们的示例集群来看看allocation inclusion 是怎么工作的。最开始，用如下的命令创建一个mastering索引。

```
curl -XPOST 'localhost:9200/mastering' -d '{ "settings" : { "index" : {
"number\_of\_shards" : 2, "number\_of\_replicas" : 0 } } }'
```

创建索引后，试着执行如下的命令：

```
curl -XPUT 'localhost:9200/mastering/\_settings' -d '{
"index.routing.allocation.include.tag": "node1", "index.routing.allocation.include.group":
"groupA", "index.routing.allocation.total\_shards\_per\_node": 1 }'
```

如果让索引状态可视化，那么集群看起来应该跟下面的图差不多.



正如你所看见的，Mastering索引的分片只分配到了tag属性值为node1或者group属性值为groupA的节点。

"结点必须"属性

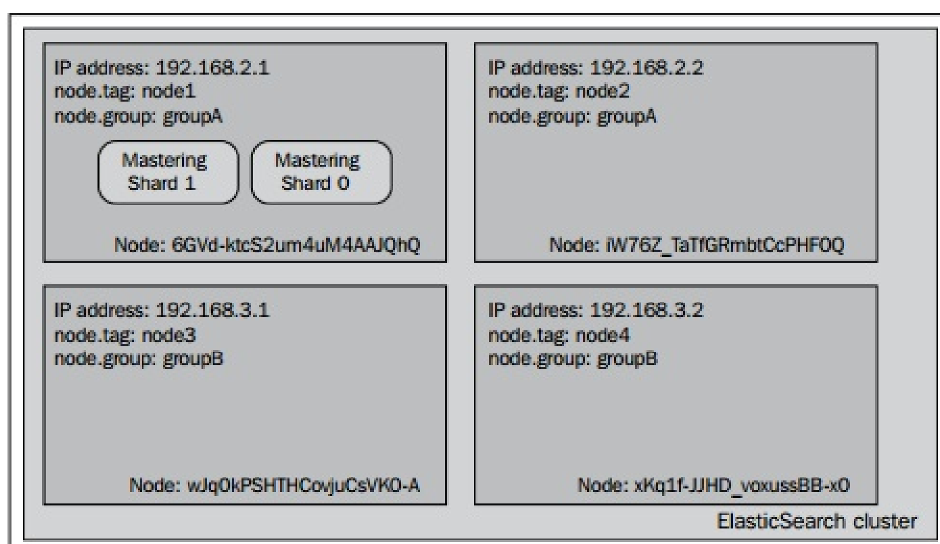
现在对我们的示例集群再回收利用(假定集群中已经没有任何索引存在)。我们再一次用如下的命令创建一个mastering索引：

```
curl -XPOST 'localhost:9200/mastering' -d '{ "settings" : { "index" : {
"number\_of\_shards" : 2, "number\_of\_replicas" : 0 } } }'
```

随后，试着执行下面命令：

```
curl -XPUT 'localhost:9200/mastering/\_settings' -d '{
"index.routing.allocation.require.tag": "node1", "index.routing.allocation.require.group":
"groupA" }'
```

如果让索引状态可视化，那么集应该跟如下图所示：



我们可以看到图示跟使用include属性有些不同。这是因为我们告诉ElasticSearch把Mastering索引的分片只分配到满足require参数所有设定值的节点上，在本例中只有第一个节点满足条件。

"结点排除"属性

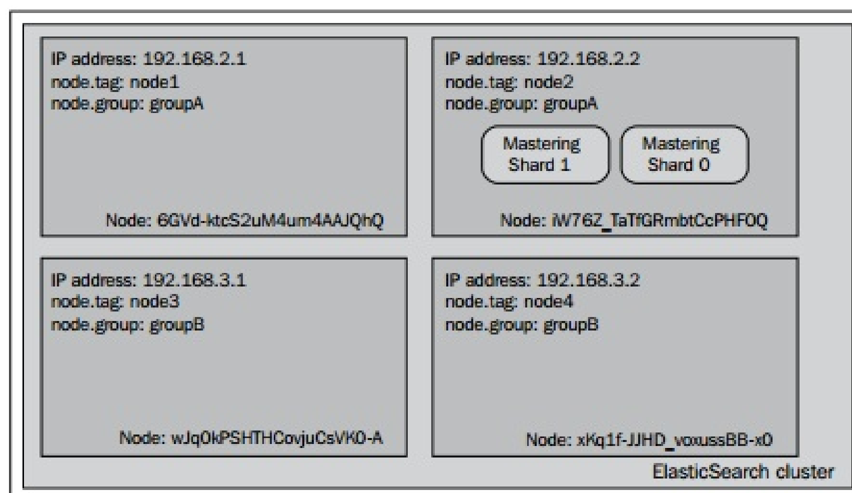
我们再一次使用示例集群，并且用如下的命令创建mastering索引：

```
curl -XPOST 'localhost:9200/mastering' -d '{ "settings" : { "index" : {
"number\_of\_shards" : 2, "number\_of\_replicas" : 0 } } }'
```

随后，试着执行下面的命令来测试allocation exclusion属性：

```
curl -XPUT 'localhost:9200/mastering/\_settings' -d '{
"index.routing.allocation.exclude.tag": "node1", "index.routing.allocation.require.group":
"groupA" }'
```

接下来，查看集群中各个节点的状态：



正如所见的那样，我们需要group属性值为groupA，但同时我们又要排除tag属性中值为node1的节点。这导致Mastering索引的分片被分配到了IP地址为192.168.2.2的节点上，这也是我们所希望的。

其它的shard allocation属性

除了前面提到的那些属性，在配置shard allocation时，ElasticSearch还提供了其它的几个特性。下面我们一起来了解一下这些属性，看看集群中还有哪些是我们可以控制的

- cluster.routing.allocation.allow_rebalance: 这个属性用来控制rebalancing发生的时间，它是基于集群中分片的状态来判断的。这个属性有以下几个可选值：[always,indice_primaries_active, indices_all_active]。如果设置属性值为always，则rebalancing操作时，不用判断集群中分片的状态。(这个值要小心使用，因为它能导致集群出现高负载状态);如果设置属性值为indice_primaries_active，当所有的主分片都可用时，rebalancing才会发生，如果设置属性值为indices_all_active，那么必须所有分片(主分片和分片副本)都已经分配就位，rebalancing才会发生。默认值是indices_all_active。
- cluster.routing.allocation.cluster_concurrent_rebalance:该属性的默认值为2，指定了集群中同一时间允许的rebalance操作的并发数。如果该值设置得比较大，将会导致比较高的I/O，比较频繁的网络活动以及比较高的节点负载。

- cluster.routing.allocation.node_initial_primaries_recoveries:该属性指定了每个节点可以同时恢复的主分片数量。由于主分片的恢复通常比较快，所以就算该值设置得比较高也不会给节点带来太大的压力。该属性的默认值是4。
- cluster.routing.allocation.node_concurrent_recoveries:该属性值默认为2。用来指定单节点上恢复操作的并发数。需要记住的是，如果值设置的过大，将到导致非常频繁的I/O活动。
- cluster.routing.allocation.disable_new_allocation:该属性值默认为flase。用来禁止新创建的索引分配分片(主分片和分片副本都算在内)。该属性可以用于以下场景：出于某些原因，希望新创建的索引暂时不进行分片的分配。该属性同时也可以用来禁止现有的索引分配新的分片，只需要在该索引中设置 index.routing.allocation.disable_new_allocation属性的值为true即可。
- cluster.routing.allocation.disable_allocation:该属性的默认值是false，用来禁止分配已经创建好的分片和分片副本。需要注意把分片副本提升成主分片(在主分片不存在时)操作并不属于分片分配，所以即使该属性值设置为true,对分片提升操作也没有影响。该属性可以用于以下场景：需要短时间禁止新创建的索引进行分片的分配。
- cluster.routing.allocation.disable_replica_allocation:该属性值默认为false，如果该属性值设置为true，分片副本的分配将会被禁止。该属性可用于以下场景：需要暂时停止分片副本的分配。该属性也可通过在索引的设置项中设置 index.routing.allocation.disable_replica_allocation为true来禁止某个特定索引的分片副本的分配。

上面提到的所有属性都是既可以在elasticsearch.yml文件中设置，也可以用update API来设置。但是在实际应用中，用户一般只使用update API来使设置生效，比如 cluster.routing.allocation.disable_new_ allocation, cluster.routing.allocation.disable_allocation, 或者 cluster.routing.allocation.disable_replica_allocation

改变分片的默认分配方式

在前面的章节中，我们学习了很多关于分片的知识以及与之相关的特性。我们也讨论了shard allocation的工作方式(本章的调整集群的分片分配一节)。然而除了默认的分配方式，我们并没有探讨其它的内容。ElasticSearch提供了更多的分片分配策略来构建先进的系统。在本节，我们将更深入地了解在分片分配方面，我们还能做哪些事情。

ShardAllocator介绍

ShardAllocator是决定分片安置到哪个节点起主要作用的一个。 当ElasticSearch改变数据在节点上的分配时，比如集群拓扑结构的改变(当节点添加或者移出集群)或者用户强制集群进行再平衡操作，ElasticSearch中分片就需要重新分配。在ElasticSearch内部，分配器继承org.elasticsearch.cluster.routing.allocation.allocator.ShardsAllocator接口。ElasticSearch提供了两种类型的分配器，它们分别是：

- even_shard
- balanced(默认实现)

我们可以通过在elasticsearch.yml文件中或者settings API设置cluster.routing.allocation.type属性来指定一种分配器的接口实现方式。

even_shard 分片分配器

早在0.90.0版本之前，ElasticSearch就已经支持该分配器。它唯一的作用就是确保每个节点中分片数量都相同(当然，这一点并不总是能得到保证的)。它也不允许把主分片和它的副本存储在同一个节点上。当需要重新调整分片分布时，如果使用even_shard,只要集群没有达到完全平衡的状态或者分片间已经无法再移动了，ElasticSearch就会把一些分片从分片密集节点移动到分片稀疏节点。需要注意的是，这种分配器的应用层面并非索引层面，这意味着只要分片和它的副本分布在不同的节点上即可，它并不关心来自同一个索引的不同分片分配到哪个节点。

balanced 分片分配器

该分配器是ElasticSearch 0.90.0版本新引入的，它是基于服务器的重要程度来分配分片,这个重要程度是用户可以掌控的。与前面提到的even_shard分配器相比，它通过暴露一些参数接口给用户来实现分片分配过程的调整，这些参数可以用集群的update API实时更新，它们是：

- cluster.routing.allocation.balance.shard:默认值 0.45
- cluster.routing.allocation.balance.index:默认值 0.5
- cluster.routing.allocation.balance.primary:默认值 0.05
- cluster.routing.allocation.balance.threshold:默认值 1.0

上述的参数决定了balanced分配器的行为。从第一个开始一一介绍，首先是基于分片数量的权重因子；其次是基于某个索引所有分片的权重因子；最后是基于主分片的权重因子。我们暂时将threshold参数放在一边，不作解说。对于某个特定的因子，其权重值越大，表明它越重要，对ElasticSearch在分片重新分配的决策上产生和影响也越大。

第一个因子用来告诉ElasticSearch每个节点上分配数量相近的分片在用户心中的重要程度。第二个因子的作用差不多，只是不针对整个集群所有的分片，针对的是整个索引的所有分片。第三个因用来告诉ElasticSearch每个节点上分配数量相同的主分片在用户心中的重要程度。所以，如果你的集群中，保证每个节点拥有相同数量的主分片这一原则非常重要，那么就应该提升cluster.routing.allocation.balance.primary的权重因子值，同时降低除threshold外其它因子的权重值。

最后，如果所有因子乘以权重后的和如果比设置的threhold值要大，那么该索引的分片就需要进行重新分配了。如果由于某个原因，你希望一些因子不影响分片的分配，那么设置其值为0即可。

自定义ShardAllocator

有些情况下，系统内置的分配器可能不适用于用户的系统部署方案。比如，在在分配分片的过程中要考虑到索引的大小；又比如，一个由不同硬件组件，处理能力不同的CPU，数量不同的内存，容量不同的硬盘组成的超大集群。所有的这些因素都可能导致分布式系统分发数据到各个节点时效率不高。 令人高兴的是，在ElasticSearch中能够实现自己的解决方案，只需要编写自己的Java类，实现org.elaticsearch.cluster.routing.allocation.allocator.ShardsAllocator接口，然后把全限定类名作为cluster.routing.allocation.type属性的参数值，配置到集群中即可。

决策者(Deciders)

为了了解分片分配器是如何决定什么时候分片会被移动，又该移向哪个节点，我们需要探究ElaticSearch的内部实现方式，这个内部实现方式称为决策者(deciders)。它们就像是人的大脑一样制定分配决策。ElasticSearch允许用户同时使用多个决策者，所有的决策者在决策时进行投票。它们遵循一致性原则，比如，如果一个决策者反对重新分配一个分片，那么这个分片就不会被移动。如下的决策者是ElasticSearch内置的，它们的决策方式一成不变，除非修改源代码。让我们来看看哪些决策者是默认的

SameShardAllocationDecider

正如它的名字一样，这个决策者不允许数据及副本(主分片和分片副本)出现在同一个节点上的状况出现。原因很明显：我们不希望备份数据和源数据放在同一个地方。说到这个决策者，我们就不得不提cluster.routing.allocation.same_shard.host属性。它控制着ElasticSearch是否关注分片所在的物理机。其默认值为false，因为多个节点可以运行在完全一样的服务器上，通过运行多虚拟机的方式。当设置它的值为true时，这个决策者就不允许把分片和它的分片副本分配到同一个物理机上。可能看起来有点奇怪，但是想想现在各种虚拟

化技术大行其道，在现代社会甚至操作系统都无法决定其运行在哪台物理机。正因为如此，最好多依靠index.routing.allocation属性家族的其它设置方式来实现这一功能，相关的内容可以从本章调整集群的分片分配一节中了解。

ShardsLimitAllocationDecider

ShardsLimitAllocationDecider确保对于给定的索引，每个节点上分片的数量不会多于设定的数量，该值设定在index.routing.allocation.total_shards_per_node属性中，可以将该属性添加在elasticsearh.yml文件中或者用update API实时更改。默认的值是-1，代表节点上分片的数量没有任何限制。需要注意，如降低该值会导致集群强制进行分片的重新分配，在集群平衡这个过程中引发额外的负载。

FilterAllocationDecider

FilterAllocationDecider用于添加分控制片分配的相关属性，即这些属性的名字都能匹配*.routing.allocation.*正则表达式。关于该决策者的工作方式，可以在本章的adjusting shard allocation一节中找到更多的信息。

ReplicaAfterPrimaryActiveAllocationDecider

该决策者使得ElasticSearch只会在主分片分配完毕后才开始分配分片副本。

ClusterRebalanceAllocationDecider

ClusterRebalanceAllocationDecider允许集群根据当前的状态改变集群再平衡的结束时间点。该决策者可以用cluster.routing.allocation.allow_rebalance属性控制，该属性有如下三种值可用：

- indices_all_active:它是默认值，表示只有集群中所有的节点分配完毕，才能认定集群再平衡完成。
- indices_primaries_active:这个值表示只要所有主分片分配完毕了，就可以认定集群再平衡完成。
- always:它表示即使当主分片和分片副本都没有分配，集群再平衡操作也是允许的。

注意这些值无法在系统运行时更改。

ConcurrentRebalanceAllocationDecider

ConcurrentRebalanceAllocationDecider能够对分片重定位操作进行限制，通过cluster.routing.allocation.cluster_concurrent_rebalance属性实现。利用该属性，我们可以设置给定集群中同时进行分片重定位的分片个数。系统默认值是2，表示集群中最多有两个分片可以同时移动。如果设置值为-1，就关闭了限制功能，这意味着rebalance的并发数不受限制。

DisableAllocationDecider

DisableAllocationDecider是另一种通过调整分片分配方式来满足业务需求的决策者，我们可以通过更改如下的设置(可以在elasticsearch.yml中静态修改或者用集群的settings API动态修改)：

- cluster.routing.allocation.disable_allocation:这个设置项用来停止所有分片的分配。
- cluster.routing.allocation.disable_new_allocation:这个设置项用来停止所有的新的主分片的分配。
- cluster.routing.allocation.disable_replica_allocation:这个设置项用来停止所有的分片副本的分配。

这些设置项都默认设置为false。当希望完全控制分片什么时候可以进行分配操作时，这些设置项用起来很方便。比如，你希望重新配置一些节点，然后重新启动使配置生效，那么就可以事先停止分片的reallocations。此外需要记住，尽管上述设置项可以在elasticsearch.yml文件中设置，但是通常用update API会更有意义。

AwarenessAllocationDecider

AwarenessAllocationDecider负责分配感知(awareness)的功能。只要使用了cluster.routing.allocation.awareness.attributes设置项，这个决策者就会生效。更多相关的功能可以参看本章调整集群的分片分配一节的内容。

ThrottlingAllocationDecider

ThrottlingAllocationDecider跟前面提到的ConcurrentRebalanceAllocationDecider有些类似。这个决策者可以用来限制allocation过程中产生的系统负载。我们可以通过如下的属性来操控这个决策者：

- cluster.routing.allocation.node_initial_primaries_recoveries:这个属性的默认值为4，它用来描述单个节点上允许recovery操作的初始主分片数量。
- cluster.routing.allocation.node_concurrent_recoveries:它的默认值是2，它用来限制单个节点上进行recovery操作的并发数。

RebalanceOnlyWhenActiveAllocationDecider

RebalanceOnlyWhenActiveAllocationDecider用来限制rebalancing过程只了生在所有分片都活跃于同一个分片复制组(主分片和它的分片副本)这一状态下。

DiskThresholdDecider

DiskThresholdDecider是在ElasticSearch 0.90.4版本引入的功能。它允许用户基于可用磁盘空间来分配分片。它默认是关闭的。如果想启动它，则需要把cluster.routing.allocation.disk.threshold_enabled属性设置为true。这个决策者可以通过配置阈值来控制何时可以将分片分配到该节点，何时ElasticSearch应该把分片分配到其它节点。

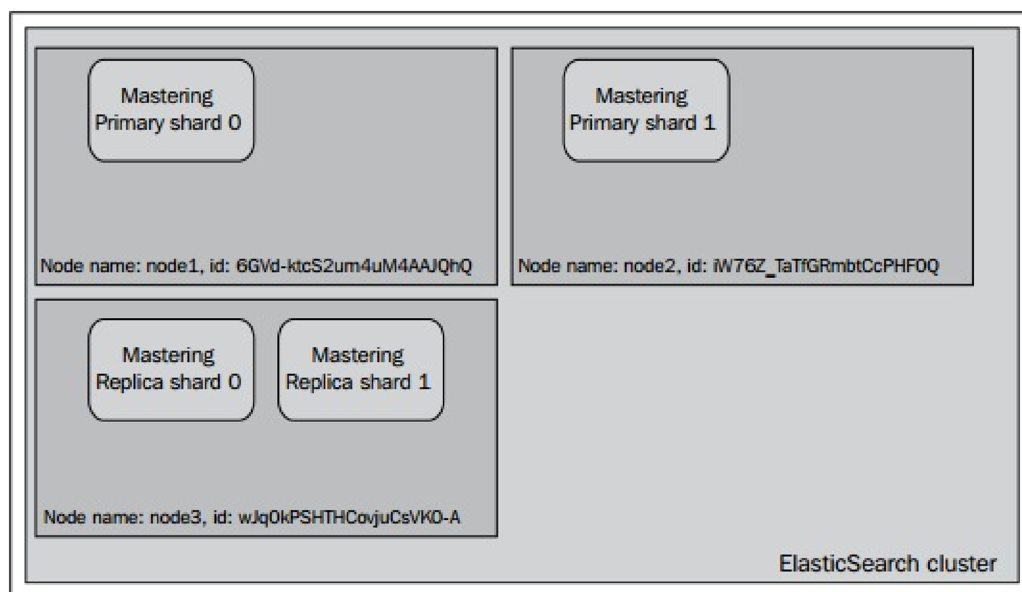
cluster.routing.allocation.disk.watermark.low属性允许用户指定一个百分比阈值或者绝对数值来控制何时能够进行分片分配。比如默认值是0.7，表示当可用磁盘空间低于70%时，新的分片才可以分配到该节点上。

cluster.routing.allocation.disk.watermark.high属性允许用户指定一个百分比阈值或者绝对数值来控制何时需要将分片分配到其它的节点。比如默认值是0.85，表示当可用磁盘空间高于85%时，ElasticSearch会重新把该节点的分片分配到其它节点。

cluster.routing.allocation.disk.watermark.low属性和cluster.routing.allocation.disk.watermark.high属性都可以指定一个百分比阈值(比如0.7或者0.85)或者绝对数值(比如1000mb)。来控制何时需要将分片分配到其它的节点。比如默认值是0.85，此外，上述属性都可以通过elasticsearch.yml静态设置或者用ElasticSearch API动态调整。

查询命令的execution preference

暂时先忘记分片分配以及分配方式，ElasticSearch不仅提供了关于分片和分片副本各式各样的设置方式，同时也提供了指定查询命令(还有其它的操作，比如real time get)执行位置的功能。在详细了解该功能之前，先看看样例集群：



通过上图可以看到，样例集群有3个节点和一个名称为**Mastering**的索引。索引拆分成两个主分片，每个分片都有一个分片副本。

preference参数简介

为控制客户端发送的查询(及其它操作)命令在集群中执行的位置，我们用到了preference参数，该参数可指定如下的值：

- _primary: 使用该属性值，发送到集群的相关操作请求只会在主分片上执行。如果发送查询命令到mastering索引时附带了值为_primary的preference参数，该命令将只在名字为node1和node2的节点上执行。比如，如果用户集群的主分片在一个机架中，分片副本在另一个机架中，用户就可能希望命令只在主分片中执行以避免使用网络流量。
- _primary_first:该属性值与_primary属性值导致相似的集群行为，但是具有容错机制。如果发送查询命令到mastering索引时附带了值为_primary_first的preference参数，该命令将在名称为node1和node2的节点上执行，但是如果有一个(或者更多)的主分片失效，查询命令将转到其它的分片上执行，在本例中会转到node3上执行。正如我们所说，该属性值与_primary属性值相似，但是如果由于某些原因主分片失效了，那么命令就会回转到分片副本上执行。
- _local: ElasticSearch会优先在本地的节点上执行相关操作。比如，如果我们向node3发送附带一条preference参数值为_local的查询命令，最终该查询命令会在node3上执行。但是，如果我们把相同的命令发送到node2节点，那么最终该命令不仅会在编号为1的分片(分片位于本地节点)上执行，同时也会分发到node1或者node3上执行，这两个

节点上有编号为0的分片。该属性值在减小网络传输时间上特别有用。只要用到了_local preference参数值，我们就能确保查询命令会尽可能地在本地的节点上执行。(比如，从本地节点运行一个客户端连接或者发送一个查询命令到节点上)

- _only_node:wJq0kPSHTHCovjuCsVK0-A:这类的操作只会在指定标识(本例中是wJq0kPSHTHCovjuCsVK0-A)的节点上执行。所以在本例中，查询命令只会在node3节点上的两个分片副本上执行。需要注意的是，如果指定节点中的分片不足以覆盖到整个索引的数据，那么命令就只会在指定节点的相关分片上执行。比如，如果我们将查询命令的preference参数值设置为 _only_node:6GVd-ktcS2um4uM4AAJQhQ，我们就只会获取到一个分片的数据。这个属性值在如下的应用场景中非常有用：用户已经知道某个节点所在的服务器处理能力强大，希望一些特定的查询命令只在该节点上执行。
- _prefer_node:wJq0kPSHTHCovjuCsVK0-A:这个选项用来把preference参数值设置成_prefer_node:，后面附带的值是一个节点Id(本例中就是wJq0kPSHTHCovjuCsVK0-A) ，这会导致ElasticSearch优先选择指定的节点来执行查询命令，但是如果该节点上缺少索引数据的一些分片，那么查询命令会发到含有欠缺分片的节点上。这与_only_node选项是类似的，_prefer_node也可以用来选择特定的节点，但是具备容错机制。
- _shards:0,1:这个preference值用来指定相关操作执行的某类分片(在本例中就是所有的分片，因为整个mastering索引只有id为0和1的分片)。这是唯一的一个可以结合其它属性使用的preference值。比如，如果希望命令只执行在本地节点的id为0或者1的分片上，我们可以把0,1两个值和_local值用“;”连接起来，最终得到的preference参数值就是这样了:0,1:_local。允许用户发出的命令只在某个分片上执行这一特性用于诊断集群问题是非常有帮助的。
- custom,字符串值:这个自定义值会确保附带相同custom值的查询命令会在同样的分片上执行。比如，如果我们的查询命令附带preference参数值为mastering_elasticsearch，那么命令会在node1和node2的主分片上执行。如果我们又发送了另一个附带同样preference参数值的查询命令，该命令也只会在node1和node2的分片上执行。该功能用于应对以下应用场景：假如集群中的各个节点刷新速率不一样，我们不希望用户在重复同一个命令时看到不同的结果，就应该使用该功能。

最后还有一点没有提到，就是ElasticSearch的默认操作。默认情况下，ElasticSearch会将相关操作随机分发到分片或者分片副本上。如果往集群中发送大量的查询命令，最终每个分片和分片副本上执行的查询命令数量会大致相同。

学以致用

随着第4章的慢慢接近尾声，我们需要获取一些接近我们日常工作的知识。因此，我们决定把一个真实的案例分成两个章节的内容。在本章节中，你将学到如何结合所学的知识，基于一些假设，构建一个容错的、可扩展的集群。由于本章主要讲配置相关的内容，我们也将聚焦集群的配置。也许结构和数据有所不同，但是对面同样的数据量集群处理检索需求的解决方案也许对你有用。

假设

在进入到纷繁的配置细节之前，我们来做一些假设，我们将基于这些假设来配置我们的ElasticSearch集群。

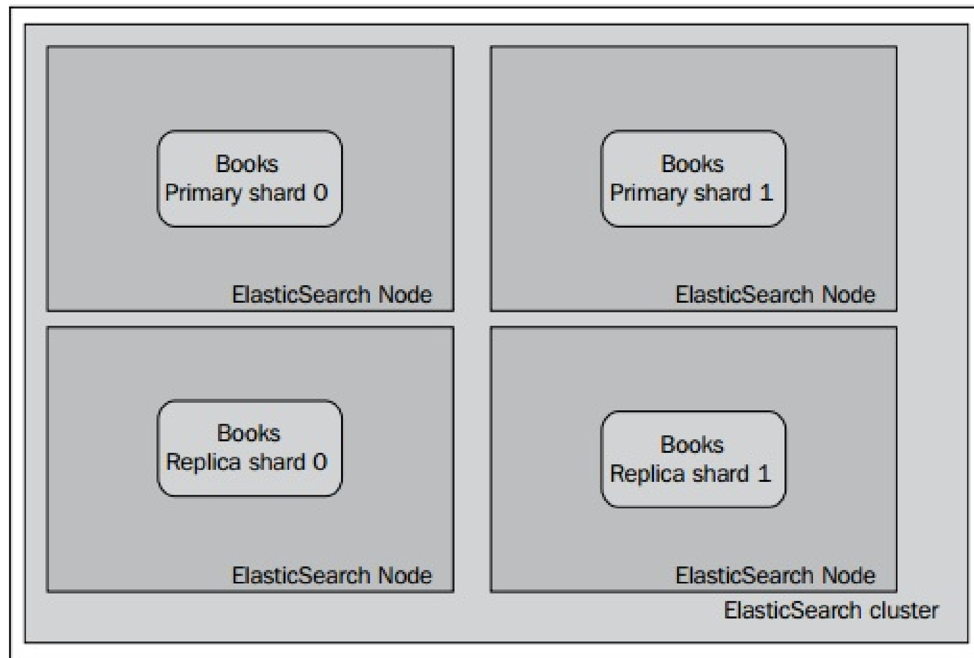
数据规模和检索性能需求

假设我们有一个在线图书馆，目前线上销售100,000种各种语言的书籍。我们希望查询请求的平均响应时间不高于200毫秒，这样就能避免用户在使用搜索服务时等待太长的时间，也能避免浏览器渲染页面时等待太长时间。所以，现在来实现期望负载。我们做了一些性能测试(内容超出本书的范围)，而且我们测到如下方案性能最好：给集群分配4个节点，数据切分到两个分片，而且每个分片挂载一个副本。



读者也许想自己做一些性能测试。如果自己做，可以选择一些开源工具来模拟用户发送查询命令到集群中。比如，Apache JMeter(http://jmeter.apache.org/)或者ActionGenerator(https://github.com/sematext/ActionGenerator) 。除此之外，还可以通过ElasticSearch提供的一些插件来查看统计记录，比如paramedic(https://github.com/karmi/elasticsearch-paramedic) ，或者BigDesk(https://github.com/lukas-vlcek/bigdesk) ，或者直接使用功能完善的监测和报警解决方案，比如Sematext公司开发，用于ElasticSearch的SPM系统(http://sematext.com/spm/elasticsearch-performancemonitoring/index.html)。所有的这些工具都会提供性能测试的图示，帮助用户找到系统的瓶颈。除了上面提到的工具，读者可能还需要监控JVM垃圾收集器的工作以及操作系统的行为(上面提到的工具中有部分工具提供了相应的功能)。

因此，我们希望我们的集群与下图类似：



当然，分片及分片副本真实的放置位置可能有所不同，但是背后的逻辑是一致的：即我们希望一节点一分片。

集群完整配置

接下来我们为集群创建配置信息，并详细讨论为什么要在集群中使用如下的属性：

```
cluster.name: books
\# node configuration
node.master: true
node.data: true
node.max\_local\_storage\_nodes: 1
\# indices configuration
index.number\_of\_shards: 2
index.number\_of\_replicas: 1
index.routing.allocation.total\_shards\_per\_node: 1
\# instance paths
path.conf: /usr/share/elasticsearch/conf
path.plugins: /usr/share/elasticsearch/plugins
path.data: /mnt/data/elasticsearch
path.work: /usr/share/elasticsearch/work
path.logs: /var/log/elasticsearch
\# swapping
bootstrap.mlockall: true
\#gateway
gateway.type: local
gateway.recover\_after\_nodes: 3
gateway.recover\_after\_time: 30s
gateway.expected\_nodes: 4
\# recovery
cluster.routing.allocation.node\_initial\_primaries\_recoveries: 1
cluster.routing.allocation.node\_concurrent\_recoveries: 1
indices.recovery.concurrent\_streams: 8
\# discovery
discovery.zen.minimum\_master\_nodes: 3
\# search and fetch logging
index.search.slowlog.threshold.query.info: 500ms
index.search.slowlog.threshold.query.debug: 100ms
index.search.slowlog.threshold.fetch.info: 1s
index.search.slowlog.threshold.fetch.debug: 200ms
\# JVM gargabe collection work logging
monitor.jvm.gc.ParNew.info: 700ms
monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

接下来了解各个属性值的意义。

节点层面的配置

在节点层面的配置中，我们指定了一个集群名字(使用cluster.name属性)来标识我们的集群。如果在同一个网段中配置了多个集群，名字相同的节点会守护甜心连接成一个集群。接下来，这个特殊的节点会被选举成主节点(用node.master:true属性)，而且该节点可以容纳索引数据(node.data:true)。此外，通过设置node.max_local_storeage_nodes属性值为1，可以限制一个节点上最多能够运行1个ElasticSearch实例。

索引的配置

由于我们只有一个索引，而且暂时也不打算添加更多的索引，我们决定设置分片的默认数量为2(用index.number_of_shards属性)，设置分片副本的默认数量为1(用index.number_of_replicas属性)。此外，我们还设置了index.routing.allocation.total_shards_per_node属性值为1，这意味着对于每个索引，ElasticSearch只会在单个节点上分配一个分片。这应用到我们的4-节点集群的例子中就是每个节点会平均分配所有的分片。

各种目录的规划

我们已经把ElasticSearch安装到了/usr/share/elasticsearch目录，基于此，conf目录、plugins目录和工作目录都在该目录下。由于这个原因，我们把数据单独指定到硬盘的一个地方，这个地方就是/mnt/data/elasticsearch挂载点。最后，我们把日志文件安置到/var/log/elasticsearch目录。基于这样的目录规划，我们在做配置的更新操作时,只需要关注/usr/share/elasticsearch目录即可，无需接触其它的目录。

Gateway的配置

正如读者所了解的，gateway是负责存储索引和元数据的模块。在本例中，我们选择推荐的，也是唯一没有废弃的gateway类型，即local（gateway.type属性）。我们说我们希望当集群只有三个节点时,恢复进程就启动(gateway.recover_after_nodes属性)，同时至少3个节点相互连接30秒后开始恢复任务(用gateway.recover_after_time属性)。此外，我们还可以通过设置gateway.expected_nodes属性值为4，用来通知ElasticSearch，我们的集群将由4个节点组成。

集群恢复机制

对于ElasticSearch来说，最核心的一种配置就是集群恢复配置。尽管它不是每天都会用到，正如你不会每天都重启ElasticSearch，也不希望集群经常失效一样。但是防范于未然是必须的。因此我们来讨论一下用到的相关属性。我们已经设置了cluster.routing.allocation.node_initial_ primaries_recoveries属性为1，这意味着我们只允许每个节点同时恢复一个主分片。这没有问题，因为每个服务器上只有一个节点。然而请记住这个操作基于gateway的local类型时会非常快，因此如果一个节点上有多个主分片时，不妨把这个值设置得大一点。 我们也设置了cluster. routing.allocation.node_concurrent_recoveries属性值为1，再一次限制每个节点同时恢复的分片数量(我们的集群中每个节点只有一个分片，

不会触发这条属性的红线，但是如果每个节点不止一个分片，而且系统I/O允许时，我们可以把这个值设置得稍微大一点)。此外，我们也设置了indices.recovery.concurrent_streams属性值为8，这是因为在最初测试recovery过程时，我们了解到我们的网络 和服务器在从对等的分片中恢复一个分片时能够轻松地使用8个并发流，这也意味着我们可以同时读取8个索引文件。

节点发现机制

在集群的discovery模块配置上，我们只需要设置一个属性：设置discovery.zen.minimum_master_nodes属性值为3。它指定了组成集群所需要的最少主节点候选节点数。这个值至少要设置成节点数的50%+1，在本例中就是3。它用来防止集群出现如下的状况：由于某些节点的失效，部分节点的网络连接会断开，并形成一个与原集群一样名字的集群(这种情况也称为“集群脑裂”状况)。这个问题非常危险，因为两个新形成的集群会同时索引和修改集群的数据。

记录慢查询日志

使用ElasticSearch时有件事情可能会很有用，那就是记录查询命令执行过程中一段时间或者更长的日志。记住这种日志并非记录命令的整个执行时间，而是单个分片上的执行时间，即命令的部分执行时间。在本例中，我们用INFO级别的日志来记录执行时间长于500毫秒的查询命令以及执行时间长于1秒的real time get请求。在调试时，我们把这些值分别设置为100毫秒和200毫秒。如下的配置片段用于上述需求：

```
index.search.slowlog.threshold.query.info: 500ms
index.search.slowlog.threshold.query.debug: 100ms
index.search.slowlog.threshold.fetch.info: 1s
index.search.slowlog.threshold.fetch.debug: 200ms
```

记录垃圾回收器的工作日志

最后，由于我们的集群没有监控解决方案(至少刚开始没有)，我们想看到垃圾收集器的工作状态。说得更清楚一点，我们希望看到垃圾回收器是否花了太多的时间，如果是，是在哪个时间段。为了实现这一需求，我们在elasticsearch.yml文件中添加下面的信息：

```
monitor.jvm.gc.ParNew.info: 700ms monitor.jvm.gc.ParNew.debug: 400ms
monitor.jvm.gc.ConcurrentMarkSweep.info: 5s
monitor.jvm.gc.ConcurrentMarkSweep.debug: 2s
```

在INFO级别的日志中，ElasticSearch会把运行时间太长的垃圾回收过程的相关信息记录下来，按照设置，阈值为 concurrent mark sweep收集器收集过程超过5秒，新生垃圾收集超过700毫秒。我们也添加了DEBUG级别的日志来应对debug需求和问题的修复。



内存设置

如果不清楚什么是新生代垃圾回收，或者不清楚什么是concurrent mark sweep，请参考Oracle的Java文档：
http://www.oracle.com/technetwork/java/javase/ gc-tuning-6-140523.html.

直到现在我们都没有提到RAM内存的设置，所以本节来[illegible]点。假设每个节点都有16GB RAM。通常不推荐将JVM 堆内存设置高于可用内存的50%，本例也是如此。我们设置Java的Xms属性值为8g，对于我们的应用来说应该够用了。由于我们的索引数据量不大，而且由于不需要facet较高基于的域，所以就没有parent-child关系型数据。在前面显示的配置信息中，我们在ElasticSearch中也设置了垃圾回收器的相关参数，但是对于长期监测，最好使用专业的监控工具，比如SPM(http://sematext.com/spm/index.html)或者Munin(http://munin-monitoring.org/)。



> 我们已经提到通用的规则，即50%的物理内存用于JVM，余下的内存用于操作系统。就像其它绝大部分规则一样，这条规则也适用于绝大部分的场景。但是我让设想一下，我们的索引数据会占到30GB的硬盘空间，我们有128GB的RAM内存，但是考虑到parent-child关系型的数据量和高基数的域中进行faceting操作，如果分配到JVM的堆内存是64G就会有出现out-of-memory异常的风险。在这样的安全中，是否依然只分配50%的可用内存空间呢？在我们看来，答案是NO，但这只适用于特殊的案例，前面提到从128G内存中JVM分配64G之后，单个索引的数据量远远小于JVM中可用内存的大小，所以我们可以适当增加。但是一定要记住给操作系统留下足够的内存以避免swapping的出现。

遗失的美好

还有一点没有提到，就是bootstrap.mlockall属性。该属性能够让ElasticSearch将堆内存锁住，并确保该块内存不会被操作系统替换成虚拟内存。如果把bootstrap.mlockall设置为true，推荐用户把ES_MIN_ME和ES_MAX_ME两个属性设置成相同的值。这样做可以确保服务器有足够的物理内存来启动ElasticSearch，并且保留足够的内存给操作系统让系统流畅运行。我们将在第6章应对突发事件 的 避免*Unix-like*操作系统的*swapping*操作一节中了解更多的相关知识。

量变引起质变

假定现在我们的服务做得很成功。访问的流量也逐步增长。而且，一家大公司希望跟我们合作。这家大的供应商不是卖自己的书，只是供货给零售商。预计该公司大概会上线200万种图书，所以系统需要处理的数据量将是现在的20倍(只估算索引文档的数量)。我们必须为这些变化作准备，也就是说要更改我们的ElasticSearch集群，使我们的用户体验能够得到保持甚至提升现。我们需要做什么呢？先解决容易的事情。我们可以更改(增加或者减少)分片副本的数量，这无需做其它的工作。这样做系统就可以同时执行更多的查询命令，当然也会相应地增加集群的压力。这样做的缺点就是增加了额外的硬盘空间开销。我们同时也要确保额外的分片副本可以分配到集群的节点上(参考选择恰当的分片数量和分片副本数量一节中的那个公式)。还要记住性能测试的结论：作为结果的吞吐量指标永远依赖于多个无法用数学公式刻画的因素。

添加主分片怎么样？前面已经提到，我们无法在线修改主分片的数量。如果我们事多分配分片，就为预期的数据增长预留了空间。但是在本例中，集群有2个主分片，应对100,000的数据足够了。但是在短时间里对于2,100,000(已经处理的数据和将要添加进来的数据)的数

据量来说太少。谁会预想到会这么成功呢？因此，必须设想一个可以处理数据增长的解决方案，但是又必须尽可能减少停服的时间，毕竟停服就意味着金钱的损失。

重新索引

第一个选择就是删除旧的索引，然后创建有更多分片的索引。这是最简单解决办法，但是在重新索引期间服务不可用。在本例中，准备用于添加到索引数据是一个耗时的过程，而且从数据库中导入数据用的时间也很长。公司的经营者说在整个重新索引数据期间停止服务是不可行的。第二个想法是创建第二个索引，并且添加数据，然后把应用接口调转到新的索引。方案可行，但是有个小问题，创建新的索引需要额外的空间开销。当然，我们将拥有新的存储空间更大的机器(我们需要索引新的“大数据”)，但是在得到机器前，我们要解决耗时的任务。我们决定寻找其它的更简单的解决方案。

路由

也许我们的例子中用routing解决会很方便？显而易见的收获就是通过routing可以用查询命令只返回我们数据集中的书籍，或者只返回属于合作伙伴的书籍(因为routing允许我们只查询部分索引)。然而，我们需要应用恰当的filter,routing不保证来自两个数据源的数据不在同一个分片上出现。不幸的是，我们的例子中还有另一个死胡同，引入routing需要进行重新索引数据。因此，我们只得把这个解决方案扔到桌子边的垃圾桶里。

多索引结构

让我们从基本的问题开始，为什么我们只需要一个索引？为什么我们要改变当前的系统。答案是我们想要搜索所有的文档，确定它们是来自于原始数据还是和作伙伴的数据。请注意ElasticSearch允许我们直接搜索多个索引。我们可以通过API端点使用多个索引，比如，/book,partner1/。我们还有一个灵巧的方法简单快速添加另一个合作伙伴，无需改变现有集群，也无需停止服务。我们可以用过别名(aliases)创建虚拟索引,这样就无需修改应用的源代码。

经过头脑风暴，我们决定选择最后一个解决方案，通过一些额外的改善使得ElasticSearch在索引数据时压力不大。我们所做的就是禁止集群的刷新率，然后删除分片副本。

> curl -XPUT localhost:9200/books/_settings -d '{ "index" : { "refresh_interval" : -1, "number_of_replicas" : 0 } }'

当然，索引数据后我们变回它原来的值，唯一的一个问题就是ElasticSearch不允许在线改变索引的名字，这导致在配置文件中修改索引名称时，会使用服务短时间停止一下。

本章小结

在本章，我们学到了在部署ElasticSearch集群时如何选择恰当数量的分片和分片副本；也了解了在索引和搜索过程中routing是如何起作用的；我们也见识了新的shard allocator是如何起作用的，也清楚了如何根据需求来配置它。我们也能够根据需求配置allocation mechanism，也学会了如何使用query execution preference功能来实现在特定的节点上执行特定的操作。最后，我们用相关知识配置了一个眞实场景的集群，并且能够依据需求进行扩展

在下一章，我们将更多地关注ElasticSearch的配置选项：我们将学习如何配置内存，如何选择合适的directory。我们将探究Gateway和Discover模块的配置，了解为什么它是如此重要。此外，我们将学习如何配置索引恢复功能，了解从Lucene的段文件中能够得到什么信息。最后将学习ElasticSearch的缓存功能。

第5章 管理Elasticsearch

前面的章节中，我们已经学习了如何为我们的应用选择正确数量的分片和分片副本，什么是分片的过度分配而且什么时候可以对分片进行过度分配。我们也深入讨论了routing机制的细节而且也了解了新引入的分片分配器是如何工作的，还有我们如何改变它的工作方式。此外，我们也学习了如何指定查询命令执行的分片。最后，我们倾其所学，配置了一个具有容错和扩展功能的集群，而且掌握了用户对应规模扩大时，集群的扩容方案。到本章结尾时，你将学到如下的知识点：

- 如何选择正确的directory实现类，使得ElasticSearch能够以最高效的方式接触到I/O系统的底层。
- 如何配置Discovery模块来避免潜在的问题
- 如何配置Gateway模块来满足我们的需求
- Recovery模块能够带给我们什么，如何修改它的配置项
- 如何查看索引段文件信息
- ElasticSearch的缓存长啥样，它负责的工作是什么，怎么使用它，怎么修改它的配置项？

选择正确的directory实现类——存储模块

在配置集群时，数据存储模块是众多模块中不需要过多关注的一个模块，但是却是非常重要的一个模块。它允许用户控制索引数据的存储方式：持久化存储(在硬盘上)或者临时存储(在内存中)。ElasticSearch中绝大部分的存储方式都会映射到合适的Apache Lucene Directory类(http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/Directory.html)。directory模块用来存取所有索引文件，因此合理配置就显得变尤为重要了。

存储类型

ElasticSearch给用户提供了4种存储类型，下面看看它们提供了什么特性以及用户该如何使用用这些特性。

简单的文件系统存储

directory类对外最简单的实现基于文件的随机读写(Java RandomAccessFile: http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html)映射到Apache Lucene的SimpleFSDirectory(http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/SimpleFSDirectory.html)。对于简单的应用来说，这种实现方式足够了。它主要的瓶颈是在文件的多线程存取时性能很差。在ElasticSearch中，通常建议使用基于新IO的系统存储来替代简单的文件系统存储。只是如果用户希望使用简单的文件系统存储，可以设置index.store.type属性值为simplefs。

新IO文件系统存储

这种存储类型使用的directory类是基于java.nio包中的FileChannel类(http://docs.oracle.com/javase/7/docs/api/java/nio/channels/FileChannel.html)实现的，该类映射到Apache Lucene的NIOFSDirectory类(http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/NIOFSDirectory.html).这种实现方式使得多个线程同时读写文件时不会出现性能下降的问题。通过设置index.store.type属性值为niofs使用该存储类型。

[需要记住的是由于Microsoft Windows操作上的JVM虚拟机有一些bug，新IO文件系统存储代码运行在Windows上时很有可能会出现一些性能问题。bug相关的信息可以参考(http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6265734.)]

MMap文件系统存储

这种存储类型使用Apache Lucene MMapDirectory实现类(http://lucene.apache.org/core/4_5_0/core/org/apache/lucene/store/MMapDirectory.html)。它使用mmap系统调用((http://en.wikipedia.org/wiki/Mmap)来读取和随机方式完成写文件操作。在进程中，它将文件映射到相同尺寸的虚拟内存地址空间中。由于没有任何的锁操作，

多线程存取索引文件时就程序就具有可伸缩性了(可伸缩性是指当增加计算资源时，程序的吞吐量或者处理能力相应的增加)。当我们使用mmap读取索引文件，在操作系统看来，该文件已经被缓存(文件会被映射到虚拟内存中)。基于这个原因，从Lucene索引中读取一个文件时，文件不必加载到操作系统的缓存中，读取速度就会快一些。这基本上就是允许Lucene，也就是ElasticSearch直接操作I/O缓存，索引文件的存取当然会快很多。值得一提的事，MMap文件系统存储最好应用在64-位操作系统环境中，如果要用在32-位的操作系统环境中，必须确保索引足够小，而且虚拟内存空间足够。用户可以通过设置index.store.type属性值为mmapfs来使用该存储类型。

内存存储

这种存储类型是几种类型中唯一不基于Apache Lucene directory实现的(当然也可以用Lucene的RAMDirectory类来实现)。内存存储类型允许用户直接把索引数据存储到内存中，所以硬盘上不会存储索引数据。记住这一点至关重要，因为这意味着数据并没有持久化：只要整个集群重启，数据就会丢失。然而，如果你的应用需要一个微型的、存取快速的，能有多个片分和分片副本的而且重建过程很快的索引，内存存储类型可能是你需要的。把index.store.type属性值设置为memeory即可使用该存储类型。

> 存储在内存中的索引数据，与其它存储类型相似，也会在允许存数据的节点上保留分片副本。

其它的属性

一旦使用内存存储类型，我们还对缓存有一定程度的控制，这些属性非常重要。请记住每个节点都需要设置如下的属性：

- cache.memory.direct:该属性的默认值为true。如果想要把内存的存储空间分配在JVM堆内存之外，就需要设定该值。一般来说，保留其默认值是个不错的选择，这样能避免堆内存出现过载情况。
- cache.memory.small_buffer_size: 该属性的默认值为1KB,内部存储结构用来存储索引段信息和删除文档的相关信息.
- cache.memory.large_buffer_size:该属性的默认值为1MB，内部存储结构用来存储除段信息和删除文档信息之外的其它信息。
- cache.memory.small_cache_size:内部内存结构用来缓存索引段信息和删除文档信息，默认值是10MB。
- cache.memory.large_cache_size:内部内存结构用来缓存除索引段信息和删除文档信息之外的其它信息，默认值是500MB。

默认存储类型

默认情况下，ElasticSearch会使用基于文件系统的存储。尽管不同的存储类型用于不同的操作系统，被选定的存储类型依然基于文件系统。比如，simplefs类用于32-位的windows操作系统；mmapfs用于Solaris操作系统和64-位的windows操作系统，niofs用于其它的操作

系统。

> 如果希望寻找来自专家对directory 实现方式使用场景的看法，请参考：Uwe Schindler写的http://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectory-on-64bit.html和Jörg Prante写的http://jprante.github.io/applications/2012/07/26/Mmap-with-Lucene.html

通常情况下，我们都会选择默认的存储类型。然而当内存容量比较大，索引也比较大的时候，考虑使用MMap文件系统存储类型是个不错的选择。这是因为使用mmap存取索引文件时，会导致索引文件缓存到操作系统的缓存中，能同时被Apache Lucene和操作系统重复使用。

节点发现模块的配置

我们已经多次提到，ElasticSearch创建的目的就是对应集群工作环境。这是跟与ElasticSearch功能类似的其它开源解决方案(比如solr)主要的不同点。其它解决方案也许同样能或难或易地应用于多节点的分布式环境，但是对对于ElasticSearch来说，工作在分布式环境就是它每天的生活。由于节点发现机制，它最大程度简化了集群的 安装和配置。

该发现机制主要基于以下假设： 集群由cluster.name设置项相同的节点自动连接而成。这就允许了同一个网段中存在多个独立的集群。自动发现机制的缺点在于：如果有人忘记改变cluster.name的设置项，无意中连接到其它的某个集群。在这种情况下，ElasticSearch可能出于重新平衡集群状态的考虑，将一些数据移动到了新加入的节点。当该节点被关闭，节点所在的集群中会有部分数据像出现魔法一样凭空消失。

Zen 发现机制

Zen 发现机制是ElasticSearch中默认的用来发现新节点的功能模块，而且集群启动后默认生效。Zen发现机制默认配置是用多播来寻找其它的节点。对于用户而言，这是一极其省事的解决方案：只需启动新的ElasticSearch节点即可，如果各个模块工作正常，该节点就会自动添加到与节点中集群名字(cluster.name)一样的集群，同时其它的节点都能感知到新节点的加入。如果节点添加不进去，你就应该检查节点的publish_host属性或者host属性的设置，来确保ElasticSearch在监听合适的网络端口。

有时由于某些原因，多播无法使用或者由于前面提到的一些原因，你不想使用它。在比较大的集群中，多播发现机制可能会产生太多不必要的流量开销，这是不使用多播的一个充分理由。在这种情况下，Zen发现机制引入了第二种发现节点的方法：单播模式。让我们在这个知识点上停留一段时间，了解这些模式的配置相关知识。

[如果想知道关于单播和多播ping方法更多的不同点，请参考:http://en.wikipedia.org/wiki/Multicastand
http://en.wikipedia.org/wiki/Unicast.]

多播

前面已经提到，这是默认的网络传输模式。当节点并非集群的一部分时(比如节点只是刚刚启动或者重启)，它会发送一个多播的ping请求到网段中，该请求只是用来通知所有能连接到节点和集群它已经准备好加入到集群中。关于多播发送，Zen发现模块暴露出如下的设置项：

- discovery.zen.ping.multicast.address(默认是所有的网络接口):属性值是接口的地址或者名称。
- discovery.zen.ping.multicast.port(默认是54328):端口用于网络通信。
- discovery.zen.ping.multicast.group(默认是:224.2.2.4):代表多播通信的消息接收地。
- discovery.zen.ping.multicast.buffer_size(默认是:2048byte):

- discovery.zen.ping.multicast.ttl(默认是3):它代表多播信息的生存时间。只要数据包通过的路由，TTL值就废弃了。通过该参数可以限制信息接收的区域。注意路由器可以指定一个类似于TTL值的阈值来确保TTL值并非唯一可以限制数据包可以跳过路由器的因素。
- discovery.zen.ping.multicast.enabled(默认值为true):设置该属性值的值为false就关闭了多播传输。如果计划使用单播传输，就应该关闭多播传输。

单播

当像前面描述的那样关闭多播，就可以安全地使用单播。当节点不是集群的一部分时(比如节点重启，启动或者由于某些错误从集群中断开)，节点会发送一个ping请求到事先设置好的地址中，来通知集群它已经准备好加入到集群中了。相关的配置项很简单，如下：

- discovery.zen.ping.unicats.hosts:该配置项代表集群中初始结点的主机列表。每个主机由名字(或者IP地址)加端口或者端口范围组成。比如，该属性值可以是如下的写法：["master1","master2:8181", "master3[80000-81000]"] ，因此用于单播节点发现的主机列表基本上不必是集群中的所有节点，因为一个节点一旦连接到集群中的一个节点，这个连接信息就会发送集群中其它所有的节点。
- discovery.zen.ping.unicats.concurrent_connects(默认值: 10):该属性指定节点发现模块能够开启的单播最大并发连接数。

主节点候选节点个数的最小化

对于节点发现模块来说，discovery.zen.minimum_master_nodes无疑是最重要的一个属性。该属性允许用户设置集群选举时主节点的候选节点数，该数值是组建集群所必须的。该属性存在的意义就是解决集群的裂脑现象。由于某些故障(比如网络故障)，网络中原来的集群分成了多个部分，每个部分的集群名字都相同，这种现象就是集群的裂脑现象。你可以想象两个同名的集群(本应该是一个)索引着不同的数据，很容易就会出现问题。因此，专家建议使用discovery.zen.minimum_master_nodes属性，并且该属性值的下限是集群节点总数的50%+1。比如，如果你们集群有9个节点，所有的节点都可以作为主结点，我们就需要设置discovery.zen.minimum_master_nodes属性值为5。即集群中至少要有5个符合选举条件的节点，才能够选择出一个主节点。

Zen发现机制的故障检测

ElasticSearch运行时会启动两个探测进程。一个进程用于从主节点向集群中其它节点发送ping请求来检测节点是否正常可用。另一个进程的工作反过来了，其它的节点向主节点发送ping请求来验证主节点是否正常且忠于职守。但是，如果我们的网络很慢或者节点分布在不同的主机，默认的配置可能显得力不从心。因此，ElasticSearch的节点发现模块开放出了如下的属性：

- discovery.zen.fd.ping_interval:该属性的默认值是1s,指定了本节点隔多久向目标结点发送一次ping请求。

- discovery.zen.fd.ping_timeout:该属性的默认值是30s,指定了节点等待ping请求的回复时间。如果节点百分百可用或者网络比较慢，可能就需要增加该属性值。
- discovery.zen.fd.ping_retries:该属性值默认为3，指定了在目标节点被认定不可用之前ping请求重试的次数。用户可以在网络丢包比较严重的网络状况下增加该属性值(或者修复网络状况)。

Amazon EC2 discovery

> 关于Amazon EC2 discovery相关内容暂时不翻译

本地Gateway

随着ElasticSearch 0.20版本发布(有些在0.19版本中)，除默认的local类型外，其它所有的gateway类型，都将废弃并且不建议用户使用，因为在未来的ElasticSearch版本中，这些类型将被移除。如果想避免出现整个数据集重新索引的情况，用户应该只使用local类型的gateway，这也是我们为什么不探讨所有其它类型gateway的原因。 local类型的gateway使用本机硬盘存储节点上的元数据、mappings数据、索引数据。为了能够使用这种gateway类型，需要服务器的硬盘有足够的空间在不使用内存缓存的情况下存储所有数据。local类型的gateway持久化数据的方式与其它gateway有所不同，为了确保在写数据的过程中，数据不丢失，它采用同步的方式来完成写数据的功能。

> 如果想设置集群使用的gateway类型，用户需要使用gateway.type属性，默认情况下该属性值为local

local类型gateway的备份

ElasticSearch直到0.90.5版本(包括该版本)都不支持存储在local类型gateway中数据的自动备份。然而，做数据备份是至关重要的，比如，如果想把集群升级到一个比较新的版本，如果升级出现了一些问题，就需要回滚操作了。为了完成上述的操作，需要执行如下的步骤：

- 停止发生在ElasticSearch集群中的数据索引操作(这可能意味着停止rivers或者任何向ElasticSearch集群发送数据的应用)
- 用Flush API刷新所有还没有提交的数据。
- 为集群中的每一个分片至少创建一个拷贝，万一出现问题，也能找回数据。当然，如果希望尽可能简单地解决问题，也可以复制整个集群中每个节点的所有数据作为备用集群。

恢复机制的配置

我们已经提到可以使用gateway来配置ElasticSearch恢复过程的行为,但是除外之外，ElasticSearch还允许用户自己配置恢复过程。在第4章 分布式索引构架 的 改变分片的默认分配方式一节中，我们已经提到一些恢复功能的配置，但是，我们认为在gateway和recovery相关的章节中讨论这些用到的属性是最合适的。

cluster-level 恢复机制的配置

绝大多数恢复机制都是定义在集群层面的，用户通过为恢复模块设置通用的规则来控制恢复模块的工作。这些设置项如下：

- indices.recovery.concurrent_streams:该属性的默认值为3，表明从数据源恢复分片数据时，允许同时打开的数据流通道。该值越大，网络层的压力也就越大，同时恢复的速度也就越快，当然恢复速度与网络使用量和总的吞吐量也有关系。
-
- indices.recovery.max_bytes_per_sec:该属性默认设置为20MB，表示分片在恢复过程中每秒传输的最大数据量。要想关闭传输量的限制，用户需要设置该属性值为0。与并发流的功能相似，该属性允许用户在恢复过程中控制网络的流量。设置该属性一个比较大的值会导致网络变得繁忙，当然恢复过程也会加快。
-
- indices.recovery.compress:该属性值默认为true，允许用户自行决定在恢复过程是否对数据进行压缩。设置该属性值为false可以降低CPU的压力，与此同时，会导致网络需要传输更多的数据。
-
- indices.recovery.file_chunk_size:该属性值指定了用于从源数据复制到分片时，每次复制的数据块的大小。默认值是512KB,同时如果indices.recovery.compress属性设置为true，数据会被压缩。
-
- indices.recovery.translog_ops:该属性值默认为1000，指定了在恢复过程中，单个请求中分片间传输的事务日志的行数。
-
- indices.recovery.translog_size: 该属性指定了从源数据分片复制事务日志数据时每次处理的数据块的大小。默认值为512KB，同时如果如果indices.recovery.compress属性设置为true，数据会被压缩。
-

> 在ElasticSearch 0.90.0版本前，曾用到indices.recovery.max_size_per_sec属性，但是在随后的版本中，该属性值被废弃了，由indices.recovery.max_bytes_per_sec属性替代。然而，如果使用0.90.0版本前的ElasticSearch，还是有必要记住这个属性。

上述的所有属性都可以通过集群的update API或者elasticsearch.yml来设置生效。

index-level 恢复机制的配置

除了前面提到的属性，还有一个可以作用于单个索引的属性。该属性可以在elasticsearch.yml文件中设置，也可以用索引更新的API来设置，它是index.recovery.initial_shards。通常情况下，当集群中还存在着不低于quorum数量的分片，并且这些分片都可进行分配时，ElasticSearch只会恢复一个特殊的分片。quorum数量是指总的分片数量加一。通过使用index.recovery.initial_shards属性时，用户可以改变ElasticSearch中quorum的实际分片数量。该属性可设置如下值：

- quorum: 该值意味着集群中至少有(分片总数的50%+1)个分片存在并且可分配。
- quorum-1:该值意味着集群中至少有(分片总数的50%-1)个分片存在并且可分配。
- full:该值意味着集群中所有的分片都必须存在并且可分配。
- full-1:该值意味着集群中至少有(分片总数-1)个分片必须存在并且可分配。
- 整数值:表示可设置为任意的整数值，比如1,2或者5，表示至少需要存在且可分配的分片数。比如，属性值为2，表示最少需要2个分片存在，同时ElasticSearch中至少需要2个分片可分配。

了解该属性值的作用总会有用上的一天，尽管在绝大多数场景中使用默认值就足够了。

索引段数据的统计

在第*3*章 索引底层控制中 控制段合并 一节中，我们探讨了调整Apache Lucene索引段合并过程来满足业务需求的可能性。此外，在第*6*章 应对突发事件的当*I/O*过于繁忙——节流功能详解一节中，我们将讨论更多功能的参数配置。然而，为了了解需要调整哪些方面，至少先得看看索引或者索引分片中索引段的结构。

段操作相关的**API**介绍

为了更深入地了解Lucene的索引段，ElasticSearch提供了段操作相关的API，通过在执行的HTTP GET请求中附带_segments REST端点，就可以调用相关的API了。比如，我们想了解集群分片中的所有的段信息，可以执行如下的命令：

```
curl -XGET 'localhost:9200/_segments'
```

如果想查看mastering索引的段信息，就应该执行如下的命令：

```
curl -XGET 'localhost:9200/mastering/_segments'
```

我们也可以同时查看多个索引的段信息，通过如下的命令即可：

```
curl -XGET 'localhost:9200/mastering,books/_segments'
```

返回信息

调用segments API的返回信息都是基于分片的。这是因为我们的索引都是由一个或多个分片(以及分片副本)组成，而且众所周知每个分片都是一个完整的Apache Lucene的物理索引。我们假定我们有一个名为mastering的索引，而且索引中有一些文档。在索引创建时，我们已经指定索引由单个分片构成，而且没有分片副本：由于只是用于测试的索引，这样做没有什么问题。我们来看一下，执行如下的命令后，索引段信息会是什么样：

```
curl -XGET 'localhost:9200/_segments?pretty'
```

返回的JSON格式信息如下(有部分删减)：


```
{ "ok" : true, "_shards" : { "total" : 1, "successful" : 1, "failed" : 0 }, "indices" : { "mastring" : { "shards" : { "0" : [ { "routing" : { "state" : "STARTED", "primary" : true, "node" : "Cz4RFYP5RnudkXzSwe-WGw" }, "num\_committed\_segments" : 1, "num\_search\_segments" : 8, "segments" : { "_0" : { "generation" : 0, "num_docs" : 62, "deleted_docs" : 0, "size" : "5.7kb", "size\_in\_bytes" : 5842, "committed" : true, "search" : true, "version" : "4.3", "compound" : false }, ... "_7" : { "generation" : 7, "num_docs" : 1, "deleted_docs" : 0, "size" : "1.4kb", "size\_in_bytes" : 1482, "committed" : false, "search" : true, "version" : "4.3", "compound" : false } } } ] } } } }
```


我们可以看到，ElasticSearch返回了大量可供分析的信息。最顶层的息是索引名称和和分片。在本例中，通过返回信息可以看到，我们有一个编号为0的分片，该分片已经启动而且

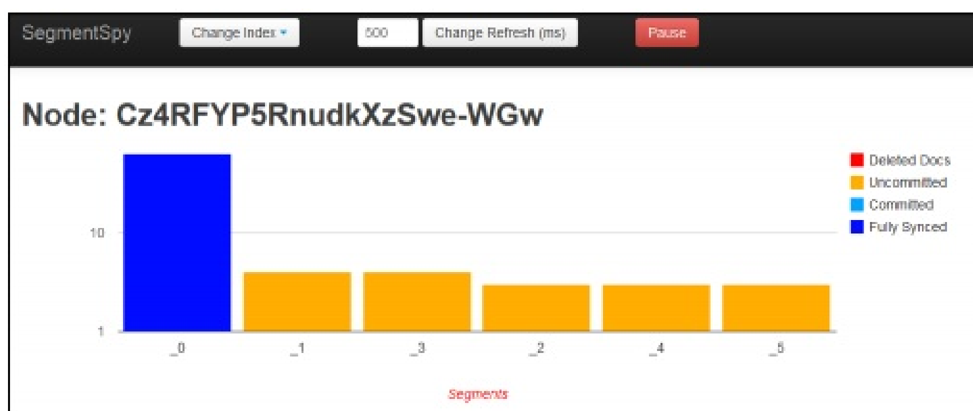
正在运行("state" : "STARTED") ，该分片是一个主分片 ("primary" : true) ,该分片位于id为Cz4RFYP5RnudkXzSwe-WGw的节点上。接下来的信息是关于已提交段的数量(通过num_commited_segments属性)和搜索段的数量(num_search_segments属性)。已提交段表示该段上面运行了一个提交的命令，即段数据已经持久化到硬盘，而且是只读的了。搜索段即可用于搜索的段。接下来是一系列的段信息，每个段中包含的信息如下：

- number:该属性为段指定了一个编号，作为多个段分组时得到的JSON对象的名称。(比如，_0,_1,等等)
- generation:该属性指定的段在索引中属于第几代，用一个整数来表示段的“老年化”程度。比如，第一个创建的段就是第0代，随后创建的段就是第1代，以此类推。
- num_docs:该属性表明该段中索引的文档数。
- deleted_docs:该属性表示该段中被标记为删除的文档数。这些标记为删除的文档将在段合并的时候被真正地删除。
- size:该属性表明段在硬盘上占用的空间大小。
- size_in_bytes:该属性表明段的大小用byte来表示时的数值。
- committed: 如果段已经提交，则该属性值为true；否则该属性值为false.
- search:该属性表示可搜索段的个数。
- version:该属性表示创建索引时使用的Lucene的版本。边注：尽管每个特定版本的ElasticSearch都只使用特定版本的Lucene，但是也是可能发生不同的索引段由不同的Lucene版本创建这一情况。像升级ElasticSearch版本，而且正好两个版本的ElasticSearch使用了不同版本的Lucene时，就会出现上面的情况。面对这种情况，旧版本的索引段会在索引合并操作时进行重写操作。
- compound:该属性指定了段的格式是否是组合的(所有的段信息都存储在一个文件)

段信息可视化

当看到segments API返回的JSON格式的文本信息时，我们大脑中冒出的第一个想法可能是：如果信息是可视化的，那就很直观明了了。如果想实现这种想法，随时可以自己去做。现在已经有一个现成的名为SegmentSpy的插件(https://github.com/polyfractal/elasticsearch-segmentspy)利用我们前面提到的API实现了段信息的可视化功能。

安装该插件后，将Web浏览器指向到http://localhost:9200/_plugin/segmentspy/ ，然后选择目标索引，我们就可以看到类似如下的屏幕截图



正如读者所见，该插件将segments API返回的信息进行了可视化的操作，这样的话想查看段信息时就能够用得上了。只是该插件没有解析ElasticSearch返回的JSON对象中的所有信息。

理解ElasticSearch缓存

缓存对于已经配置好且正常工作的集群来说不过过多关注(这一条不仅适用于ElasticSearch)。缓存在ElasticSearch中扮演着重要的角色。通过缓存用户可以高效地存储过滤器的数据并且重复使用这些数据，比如高效地处理父子关系数据、faceting、对数据以索引中某个域来排序等等。在本节中，我们将详细研究filter cache和field data cache这些最重要的缓存，而且我们将会意识到理解缓存的工作原理对于集群的调优非常重要。

过滤器缓存

过滤器缓存是负责缓存查询语句中过滤器的结果数据。比如，让我们来看如下的查询语句：

> { "query" : { "filtered" : { "query" : { "match_all" : {} }, "filter" : { "term" : { "category" : "romance" } } } } }

执行该查询语句将返回所有category域中含有term值为romcance的文档。正如读者如看到的那样，我们将match_all查询类型与过滤器结合使用。现在，执行一次查询语句后，每个有同样过滤条件的查询语句都会重复使用缓存中的数据，从而节约了宝贵的I/O和CPU资源。

过滤器缓存的类型

在ElasticSearch中过滤器缓存有两种类型：索引级别和节点层面级别的缓存。所以基本上我们自己就可以配置过滤器缓存依赖于某个索引或者一个节点(节点是默认设置)。由于我们无法时时刻刻来猜测具体的某个索引会分配到哪个地方(实际上分配的是分片和分片副本)，也就无法预测内存的使用，因此不建议使用基于索引的过滤器。

index-level过滤器缓存的配置

ElasticSearch允许用户使用如下的属性来配置index-level过滤器缓存的行为：

- index.cache.filter.type:该属性用于指定缓存的类型，有resident,soft和weak，node(默认值)4个值可供选择。对于resident类型的缓存，JVM无法删除其中的缓存项，除非用户来删除(通过API,设置缓存的最大容量及失效时间都可以对缓存项进行删除)。推荐使用该缓存类型(因为填充过滤器缓存开销很大)。soft和weak类型的缓存能够在内存不足时，由JVM自动清除。当JVM清理缓存时，其操作会根据缓存类型而有所不同。它会首先清理引用比较弱的缓存项，然后才会是使用软引用的缓存项。node属性表示缓存将在节点层面进行控制(参考本章的*Node-level*过滤器缓存配置一节的内容)
- index.cache.filter.max_size: 该属性指定了缓存可以存储缓存项的数量(默认值是-1，表示数量没有限制)。读者需要记住，该设置项不适用于整个索引，只适用于索引分片上的一个段。所以缓存的内存使用量会因索引中分片(以及分片副本)的数量，还有索引中段的数量的不同而不同。通常情况下，默认没有容量限制的缓存适用于soft类型和致

力于缓存重用的特定查询类型。

- index.cache.filter.expire:该属性指定了过滤器缓存中缓存项的失效时间，默认是永久不失效(所以其值设为-1)。如果希望一段时间类没有命中的缓存项失效，缓存项沉寂的最大时间。比如，如果希望缓存项在60分钟类没有命中就失效，设置该属性值为60m即可。



想了解更多关于软引用和虚引用相关的内容，可以参考Java Document，特别是以下两类：
http://docs.oracle.com/javase/7/docs/api/java/lang/ref/SoftReference.html 和
http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html.

Node-level过滤器缓存的配置

让缓存作用于节点上，是ElasticSearch默认和推荐的设置。对于特定节点的所有分片，该设置都已经默认生效(设置index.chache.filter.type属性值为node,或者对此不作任何设置)。ElsticSearch允许用户通过indices.cache.filter.size属性来配置缓存的大小。用户可以使用百分数，比如20%(默认设置)或者具体的值，比如1024mb来指定缓存的大小。如果使用百分数，那么ElasticSearch会基于节点的heap 内存值来计算出实际的大小。

node-level过滤器缓存是LRU类型的缓存(最近最少使用)，即需要移除缓存项来为新的缓存项腾出位置时，最长时间没有命中的缓存项将被移除。

域数据缓存

当查询命令中用到faceting功能或者指定域排序功能时，域数据缓存就会用到。使用该缓存时，ElasticSearch所做的就是将指定域的所有取值加载到内存中，通过这一步，ElasticSearch就可以提供文档域快速取值的功能。有两点需要记住：直接从硬盘上读取时，域的取值开销很大，这是因为加载整个域的数据到内存不仅需要I/O操作，还需要CPU资源。



读者需要记住，对于每个用来进行faceting操作或者排序操作的域，域的所有取值都要加载到内存中：一个Term都不能放过。这个过程开销很大，特别是对于基数比较大的域，这种域的term对象数目巨大。

index-level过滤器缓存的配置

与index-level过滤器缓存类似，我们也可以使用index-level域数据缓存，但是我们再说一次，不推荐使用index-level的缓存，原因还是一样的：哪个分片或者哪个索引分配到哪个节点是很难预测的。因此我们也无法预测每个索引使用到的内存有多少，这样容易出现内存溢出问题。

然而，如果用户用户熟知系统底层，熟知业务特点，了解resident和soft域数据缓存，可以将index.fielddata.cache.type属性值为resident或者soft来启用index-level的域数据缓存。在前面的过滤器缓存中已经有过描述，resident属性的缓存无法由JVM自动移除，除非用户介入。如果使用index-level域数据缓存，推荐使用resident类型。重建域数据缓存开销巨大，而且会影响搜索性能。soft类型的域数据缓存可以在内存不足时由JVM自动移除。

Node-level过滤器缓存的配置

ElasticSearch 0.90.0版本允许用户使用使用如下属性来设置node-level域数据缓存，如果用户没有修改配置，node-level域数据缓存是默认的类型。

- indices.fielddata.cache.size:该属性有来指定域数据缓存的大小，可以使用百分数比如20%或者具体的数值，比如10gb。如果使用百分数，ElasticSearch会根据节点的最大堆内存值(heap memory)将百分数换算成具体的数值。默认情况下，域数据缓存的大小是没有限制的。
- indices.fielddata.cache.expire:该属性用来设置域数据缓存中缓存项的失效时间，默认值是-1，表示缓存项不会失效。如果希望缓存项在指定时间内不命中就失效的话，可以设置缓存项沉寂的最大时间。比如，如果希望缓存项60分钟内不命中就失效的话，就设置该属性值为60m.



如果想确保ElasticSearch应用node-level域数据缓存，用户可以设置index.fielddata.cache.type属性值为node，或者根本不设置该属性的值即可。

域数据过滤

除了前面提到的配置项，ElasticSearch还允许用户选择域数据加载到域数据缓存中。这在一些场景中很有用，特别是用户记得在排序和faceting时使用域缓存来计算结果。ElasticSearch允许用户使用两种类型过滤加载的域数据：通过词频，通过正则表达式，或者结合这两者。

样例之一就是faceting功能：用户可能想把频率比较低的term排除在faceting的结果之外，这时，域数据过滤就很有用了。比如，我们知道在索引中有一些term有拼写检查的错误，当然这些term的基数都比较低。我们不想因此影响faceting功能的计算，因此只能从数据集中移除他们：要么从从数据源中更改过来，要么通过过滤器从域数据缓存中去除。通过过滤，不仅仅是从ElasticSearch返回的结果中排除了这些数据，同时降低了内存的占用，因为过滤后存储在内存中的数据会更少。接下来了解一下过滤功能。

添加域数据过滤的信息

为了引入域数据过滤信息，我们需要在mappings域定义中添加额外的对象：fielddata对象以及它的子对象，filter。因此，以抽象的tag域为例，扩展后域的定义如下：

```
"tag" : { "type" : "string", "index" : "not_analyzed", "fielddata" : { "filter" : { ... } } }
```

在接下来的一节中，我们将了解filter对象内部的秘密

通过词频过滤

词频过滤功能允许用户加载频率高于指定最小值(min参数)并且低于指定最大值(max参数)的term。绑定到词频的min和max参数不是基于整个索引的，而是索引的每个段，这一点非常重要，因为不同的段词频会有不同。min和max参数可以设定成一个百分数(比如百分之一就是0.01，百分之五十就是0.5)或者设定为一个具体的数值。

此外，用户还可以设定min_segment_size属性值，用于指定一个段应该包含的最小文档数。这构建域数据缓存时，低于该值的段将不考虑加载到缓存中。

比如，如果我们只想把满足如下条件的term加载到域数据缓存中：1、段中的文档数不少于100；2、段中词率在1%到20%之间。那么域就可以定义如下：

> { "book" : { "properties" : { "tag" : { "type" : "string", "index" : "not_analyzed", "fielddata" : { "filter" : { "frequency" : { "min" : 0.01, "max" : 0.2, "min_segment_size" : 100 } } } } } } }

通过正则表达式过滤

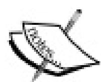
除了可以通过词频过滤，还可以通过正则表达式过滤。比如有这样的应用场景：只有符合正则表达式的term才可以加载到缓存中。比如，我们只想把tag域中可能是Twitter标签(以#字符开头)的term加载到缓存中，我们的mappings就应该定义如下：

> { "book" : { "properties" : { "tag" : { "type" : "string", "index" : "not_analyzed", "fielddata" : { "filter" : { "regex" : "^\#.*" } } } } } }

通过正则表达式和词频共同过滤

理所当然，我们可以将上述的两种过滤方法结合使用。因此，如果我们希望域数据缓存中tag域中存储满足如下条件的数据： 1、以#字符开头；2、段中至少有100个文档；3、基于段的词频介于1%和20%之间，我们应该定义如下的mappings:

> { "book" : { "properties" : { "tag" : { "type" : "string", "index" : "not_analyzed", "fielddata" : { "filter" : { "frequency" : { "min" : 0.1, "max" : 0.2, "min_segment_size" : 100 }, "regex" : "^#.*" } } } } } }



请记住域缓存不是在索引过程中构建的，因此可以在查询过程中重新构建，基于此，我们可以在系统运行过程中通过mappingsAPI更新fielddata部分的设置，然而，读者需要记住更新域数据加载过滤设置项后，缓存必须用相关的API清空。关于缓存清理API，可以在本章的清空缓存一节中了解到。

过滤功能的一个例子

接下来我们回到过滤章节开头的例子。我们希望排除faceting结果集中词频最低的term。在本例中，词频最低即频率低于50%的term，当然这个频率已经相当高了，只是我们的例子中只有4个文档。在真实产品中，你可能需要将词频设置得更低。为了实现这一功能，我们用如下的命令创建一个books索引：

> curl -XPOST 'localhost:9200/books' -d '{ "settings" : { "number_of_shards" : 1, "number_of_replicas" : 0 }, "mappings" : { "book" : { "properties" : { "tag" : { "type" : "string", "index" : "not_analyzed", "fielddata" : { "filter" : { "frequency" : { "min" : 0.5, "max" : 0.99 } } } } } } } }'

接下来，通过批处理API添加一些样例文档：

> curl -s -XPOST 'localhost:9200/_bulk' --data-binary ' { "index": {"_index": "books", "_type": "book", "_id": "1"}} {"tag":["one"]} { "index": {"_index": "books", "_type": "book", "_id": "2"}} {"tag":["one"]} { "index": {"_index": "books", "_type": "book", "_id": "3"}} {"tag":["one"]} { "index": {"_index": "books", "_type": "book", "_id": "4"}} {"tag": ["four"]} '

接下来，运行一个查询命令来检测一个简单的faceting功能(前面已经介绍了域数据缓存的操作方法)：

> curl -XGET 'localhost:9200/books/_search?pretty' -d ' { "query" : { "match_all" : {} }, "facets" : { "tag" : { "terms" : { "field" : "tag" } } } }'

前面查询语句的返回结果如下：

> { "took" : 2, "timed_out" : false, "_shards" : { "total" : 1, "successful" : 1, "failed" : 0 }, . . . "facets" : { "tag" : { "_type" : "terms", "missing" : 1, "total" : 3, "other" : 0, "terms" : [{ "term" : "one", "count" : 3 }] } } }

可以看到，term faceting功能只计算了值为one的term,值为four的term忽略了。如果我们假定值为four的term拼写错误，那么我们的目的就达到了。

缓存的清空

前面已经提到过，如果更改了域数据缓存的设置，在更新后清空缓存是至关重要的。同时，想更新一些用到确定缓存项的查询语句，清除缓存功能也是很有用的。ElasticSearch允许用户通过_cache这个rest端点来清空缓存。该rest端点的使用方法随后介绍。

单个索引、多个索引、整个集群缓存的清空

我们能做的最简单的事就是通过如下的命令清空整个集群的缓存：

> curl -XPOST 'localhost:9200/_cache/clear'

当然，我们也可以选择清空一个或者多个索引的缓存。比如，如果想清空mastering索引的缓存，应该运行如下的命令：

> curl -XPOST 'localhost:9200/mastering/_cache/clear'

同时，如果想清空mastering和books索引的缓存，应该运行如下的命令：

> curl -XPOST 'localhost:9200/mastering,books/_cache/clear'

清除指定类型的缓存

除了前面提到的缓存清理方法，我们也可以只清理指定类型的缓存。可以清空如下类型的缓存：

- filter:设置filter参数为true，该类型的缓存即可被清除。如果不希望此类型的缓存被清除，设置filter参数值为false即可。
- field_data:设置field_data参数值为true，该类型的缓存即可被清除。如果不希望此类型的缓存被清除，设置field_data参数值为false即可。
- bloom:如果想清除bloom缓存(用于倒排表的布隆过滤器，在第3章 索引底层控制的使用*Codecs*一节中有介绍)，bloom参数值应该设置为true。。如果不希望此类型的缓存被清除，设置bloom参数值为false即可

例如，如果我们想清空mastering索引中的域数据缓存，同时保留过滤器缓存和没有接触到bloom缓存，运行如下的命令即可：

> curl -XPOST 'localhost:9200/mastering/_cache/clear?field_data=true&filter =false&bloom=false'

清除域相关的缓存

除了可以清空所有的缓存，以及指定的缓存，我们还可以清除指定域的缓存。为了实现这一功能，我们需要在请求命令中添加fields参数，参数值为我们想清空的域,多个域用逗号隔开。例如，如果我们想清空mastering索引中title域和price域的缓存，运行如下的命令即可：

> curl -XPOST 'localhost:9200/mastering/_cache/clear?fields=title,price'

本章小节

在本章中，我们已经学习了如何选择正确的directory实现类来让ElasticSearch以最高效的方式进行I/O操作。我们也已经了解了如何用多播和单播方法配置节点的发现模块。我们也探讨了gateway模块，它能让我们在集群对恢复的过程进行控制，当然我们也研究了恢复模块和它的配置。此外，我们还学习了如何分析ElasticSearch返回的索引段的信息。最后，我们深入学习了ElasticSearch缓存的工作原理，学习了如何修改缓存的配置，学习了控制域数组缓存的构建。

在下一章中，我们将学习应对突发事件：我们将学习处理集群的故障。我们将首先学习Java垃圾回收器的工作，学习如何监控垃圾收集器的工作，学习分析JVM提供的信息。此外，我们将学习throttling，它能够控制ElasticSearch以及底层的Apache Lucene工具包带给I/O子系统的压力。我们也将了解warmers带给查询性能的影响，也将学习如何使用warmer。最后，我们将学习如何使用ElasticSearch提供的hot threads API，以及如何使用ElasticSearch API提供有用信息和统计结果来诊断和应对系统问题。

第6章 应对突发事件

第7章 优化用户体验

第8章 ElasticSearch Java API

第9章 开发ElasticSearch插件